

O'REILLY®



Python Fluente

PROGRAMAÇÃO CLARA, CONCISA E EFICIENTE

novatec

Luciano Ramalho

FOTOS ORIGINAIS DESSE EBOOK

https://lbry.tv/python_fluente_2015_impar

https://lbry.tv/python_fluente_2015_par

QUER SER MAIS LIVRE?

Gigantes portais mundiais de compartilhamento de ebooks e artigos científicos, com mais de 2 milhões de documentos.

Acesse:

<http://libgen.is/>

<https://b-ok.lat/>

Grupo de Telegram com mais de 70 mil ebooks em português.

Acesse:

<https://t.me/REVISTAVIRTUALBR>

Plataforma descentralizada de reprodução e hospedagem de arquivos garantida por blockchain.

- Prática como youtube
- Descentralizada e à prova de censura como torrent
- Facilitadora de transações financeiras como bitcoin

Acesse:

<http://lbry.tv/>

Livro: Contra a Propriedade Intelectual

Acesse:

<https://www.mises.org.br/Ebook.aspx?id=29>

Python Fluente

Python Fluente

Luciano Ramalho

Para Marta, com todo o meu amor.

Authorized Portuguese translation of the English edition of titled *Fluent Python*, ISBN 9781491946008 © 2015 Luciano Ramalho. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra *Fluent Python*, ISBN 9781491946008 © 2015 Luciano Ramalho. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., que detém todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. 2015.

Todos os direitos reservados e protegidos pela Lei 9610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Assistente editorial: Priscila A. Yoshimatsu

Tradução: Lúcia A. Kinoshita

Revisão técnica da tradução: Luciano Ramalho

Revisão gramatical: Marta Almeida de Sá

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-462-5

Histórico de impressões:

Junho/2020	Quinta reimpressão
Outubro/2015	Primeira edição

Novatec Editora Ltda.

Rua Luis Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

CRA20200603

Prefácio 17

Parte I ■ Prólogo..... 27

Capítulo 1 ■ Modelo de dados do Python.....28

Um baralho pythônico 29

Como os métodos especiais são usados..... 33

 Emulando tipos numéricos 34

 Representação em string..... 36

 Operadores aritméticos 37

 Valor booleano de um tipo definido pelo usuário 37

Visão geral dos métodos especiais 38

Por que len não é um método? 39

Resumo do capítulo 40

Leituras complementares..... 41

Parte II ■ Estruturas de dados..... 43

Capítulo 2 ■ Uma coleção de sequências.....44

Visão geral das sequências embutidas 45

List comprehensions e expressões geradoras 46

 List comprehensions e legibilidade..... 46

 Comparação entre listcomps e map/filter 48

 Produtos cartesianos 49

 Expressões geradoras 50

Tuplas não são apenas listas imutáveis..... 52

 Tuplas como registros..... 52

 Desempacotamento de tuplas 53

 Desempacotamento de tuplas aninhadas..... 55

 Tuplas nomeadas..... 56

 Tuplas como listas imutáveis..... 58

Fatiamento	59
Por que as fatias e os intervalos excluem o último item	59
Objetos slice.....	60
Fatiamento multidimensional e reticências.....	62
Atribuição de valores a fatias	62
Usando + e * com sequências.....	63
Criando listas de listas	64
Atribuições combinadas e sequências	65
O enigma da atribuição +=.....	67
list.sort e a função embutida sorted	69
Administrando sequências ordenadas com bisect.....	71
Pesquisando com bisect.....	71
Inserção com bisect.insort	74
Quando uma lista não é a resposta	75
Arrays.....	75
Memory Views.....	78
NumPy e SciPy	80
Deque e outras filas	82
Resumo do capítulo	86
Leituras complementares.....	87
Capítulo 3 ■ Dicionários e conjuntos.....	93
Tipos genéricos de mapeamento	94
dict comprehensions	96
Visão geral dos métodos comuns a mapeamentos.....	97
Tratando chaves ausentes com.setdefault	98
Mapeamentos com consulta de chave flexível.....	101
defaultdict: outra abordagem para chaves ausentes.....	101
Método __missing__	102
Variações de dict.....	105
Criando subclasses de UserDict.....	106
Mapeamentos imutáveis.....	108
Teoria dos conjuntos	109
Literais de set	111
Set comprehensions.....	113
Operações de conjuntos	113
Por dentro de dict e set	116
Um experimento para testar o desempenho	116
Tabelas hash em dicionários	118
Consequências práticas de como os dicionários funcionam	121
Como os conjuntos funcionam – consequências práticas	125
Resumo do capítulo	125
Leituras complementares.....	126

Capítulo 4 ■ Texto versus bytes	129
Falhas de caracteres	130
O essencial sobre bytes	131
Structs e memory views	134
Codificadores/decodificadores básicos	135
Entendendo os problemas de codificação/decodificação	137
Lidando com UnicodeEncodeError	138
Lidando com UnicodeDecodeError	139
SyntaxError ao carregar módulos com codificação inesperada	140
Como descobrir a codificação de uma sequência de bytes	142
BOM: um gremlin útil	142
Lidando com arquivos-texto	144
Defaults de codificação: um hospício	147
Normalizando Unicode para comparações mais seguras	150
Case folding	153
Funções utilitárias para comparações normalizadas	154
“Normalização” extrema: removendo acentos	155
Ordenação de texto Unicode	159
Ordenação com o Unicode Collation Algorithm	161
Base de dados Unicode	161
APIs de modo dual para str e bytes	163
str versus bytes em expressões regulares	163
str versus bytes em funções de os	165
Resumo do capítulo	167
Leituras complementares	169
 Parte III ■ Funções como objetos	 174
Capítulo 5 ■ Funções de primeira classe	175
Tratando uma função como um objeto	176
Funções de ordem superior	177
Substitutos modernos para map, filter e reduce	178
Funções anônimas	180
As sete variações de objetos invocáveis	181
Tipos invocáveis definidos pelo usuário	182
Introspecção de função	183
De parâmetros posicionais a parâmetros exclusivamente nomeados	185
Obtendo informações sobre parâmetros	187
Anotações de função	192
Pacotes para programação funcional	194
Módulo operator	194
Congelando argumentos com functools.partial	198

Resumo do capítulo	200
Leituras complementares.....	201
Capítulo 6 ■ Padrões de projeto com funções de primeira classe.....	205
Estudo de caso: refatorando Strategy	206
Strategy clássico	206
Strategy orientado a função	210
Escolhendo a melhor estratégia: abordagem simples.....	213
Encontrando estratégias em um módulo	214
Command	216
Resumo do capítulo	218
Leituras complementares.....	219
Capítulo 7 ■ Decoradores de função e closures.....	222
Básico sobre decoradores.....	223
Quando Python executa os decoradores	224
Padrão Strategy melhorado com decorador	226
Regras para escopo de variáveis.....	228
Closures	232
Declaração nonlocal.....	235
Implementando um decorador simples	237
Funcionamento	238
Decoradores da biblioteca-padrão.....	240
Memoização com <code>functools.lru_cache</code>	240
Funções genéricas com <code>dispatch</code> simples	243
Decoradores empilhados	246
Decoradores parametrizados	247
Um decorador de registro parametrizado	247
Decorador clock parametrizado	249
Resumo do capítulo	252
Leituras complementares.....	253
Parte IV ■ Práticas de orientação a objetos	257
Capítulo 8 ■ Referências a objetos, mutabilidade e reciclagem	258
Variáveis não são caixas	259
Identidade, igualdade e apelidos	260
Escolhendo entre <code>==</code> e <code>is</code>	262
A relativa imutabilidade das tuplas	263
Cópias são rasas por padrão	264
Cópias profundas e rasas de objetos quaisquer.....	267

Parâmetros de função como referências 268

 Tipos mutáveis como default de parâmetros: péssima ideia..... 270

 Programação defensiva com parâmetros mutáveis 272

del e coleta de lixo..... 274

Referências fracas..... 276

 Esquete com WeakValueDictionary 277

 Limitações das referências fracas 279

Truques de Python com imutáveis 280

Resumo do capítulo 282

Leituras complementares..... 283

Capítulo 9 ■ Um objeto pythônico 288

Representações de objetos 289

Retorno da classe Vector 289

Um construtor alternativo 292

classmethod versus staticmethod..... 293

Apresentações formatadas..... 295

Um Vector2d hashable 298

Atributos privados e “protegidos” em Python..... 304

Economizando espaço com o atributo de classe __slots__ 307

 Os problemas com __slots__ 309

Sobrescrita de atributos de classe..... 310

Resumo do capítulo 312

Leituras complementares..... 313

Capítulo 10 ■ Hackeando e fatiando sequências..... 318

Vector: um tipo de sequência definido pelo usuário..... 319

Vector tomada #1: compatível com Vector2d 319

Protocolos e duck typing..... 322

Vector tomada #2: uma sequência que permite fatiamento..... 323

 Como funciona o fatiamento 324

 Um __getitem__ que considera fatias 327

Vector tomada #3: acesso dinâmico a atributos 328

Vector tomada #4: hashing e um == mais rápido..... 332

Vector tomada #5: formatação 338

Resumo do capítulo 345

Leituras complementares..... 347

Capítulo 11 ■ Interfaces: de protocolos a ABCs 352

Interfaces e protocolos na cultura de Python..... 353

Python curte sequências..... 355

Monkey-patching para implementar um protocolo em tempo de execução..... 357

Aves aquáticas de Alex Martelli 359

Criando subclasses de uma ABC	365
ABCs da biblioteca-padrão	367
ABCs em collections.abc	367
A torre numérica de ABCs	369
Definindo e usando uma ABC	370
Detalhes de sintaxe das ABCs	374
Herdando da ABC Tombola	375
Uma subclasse virtual de Tombola	378
Como as subclasses de Tombola foram testadas	381
Uso de register na prática	384
Gansos podem se comportar como patos	385
Resumo do capítulo	386
Leituras complementares	389
Capítulo 12 ■ Herança: para o bem ou para o mal	395
Artimanhas da criação de subclasses de tipos embutidos	396
Herança múltipla e ordem de resolução de métodos	399
Herança múltipla no mundo real	404
Lidando com herança múltipla	407
1. Faça a distinção entre herança de interface e herança de implementação	407
2. Deixe as interfaces explícitas com ABCs	407
3. Use mixins para reutilização de código	407
4. Explícite as mixins pelo nome	408
5. Uma ABC também pode ser uma mixin; o contrário não é verdade	408
6. Não herde de mais de uma classe concreta	408
7. Ofereça classes agregadas aos usuários	409
8. “Prefira composição de objetos à herança de classe.”	409
Tkinter: o bom, o ruim e o feio	410
Um exemplo moderno: mixins em views genéricas de Django	411
Resumo do capítulo	414
Leituras complementares	415
Capítulo 13 ■ Sobrecarga de operadores: o jeito certo	419
Básico da sobrecarga de operadores	420
Operadores unários	420
Sobrecarregando + para soma de vetores	423
Sobrecarregando * para multiplicação por escalar	429
Operadores de comparação rica	433
Operadores de atribuição combinada	438
Resumo do capítulo	443
Leituras complementares	444

Parte V ■ Controle de fluxo	449
Capítulo 14 ■ Iteráveis, iteradores e geradores	450
Sentence tomada #1: uma sequência de palavras.....	451
Por que sequências são iteráveis: a função iter	453
Iteráveis versus iteradores	455
Sentence tomada #2: um iterador clássico.....	459
Fazer de Sentence um iterador: péssima ideia	460
Sentence tomada #3: uma função geradora.....	461
Como funciona uma função geradora	462
Sentence tomada #4: uma implementação lazy	466
Sentence tomada #5: uma expressão geradora	467
Expressões geradoras: quando usá-las	469
Outro exemplo: gerador de progressão aritmética	470
Progressão aritmética com itertools	473
Funções geradoras na biblioteca-padrão.....	474
Nova sintaxe em Python 3.3: yield from.....	485
Funções de redução de iteráveis.....	486
Uma visão mais detalhada da função iter	488
Estudo de caso: geradores em um utilitário para conversão de banco de dados	489
Geradores como corrotinas	491
Resumo do capítulo	492
Leituras complementares.....	493
Capítulo 15 ■ Gerenciadores de contexto e blocos else	499
Faça isso, então aquilo: blocos else além de if.....	500
Gerenciadores de contexto e blocos with	502
Utilitários de contextlib	507
Usando @contextmanager	508
Resumo do capítulo	512
Leituras complementares.....	512
Capítulo 16 ■ Corrotinas	515
Como as corrotinas evoluíram a partir de geradores.....	516
Comportamento básico de um gerador usado como corrotina	517
Exemplo: corrotina para calcular uma média cumulativa	520
Decoradores para preparação de corrotinas.....	522
Término de corrotinas e tratamento de exceção	524
Devolvendo um valor a partir de uma corrotina	528
Usando yield from.....	530
O significado de yield from.....	536

Caso de uso: corrotinas para uma simulação de eventos discretos	543
Sobre simulações de eventos discretos.....	543
A simulação da frota de táxis.....	544
Resumo do capítulo	552
Leituras complementares.....	554
Capítulo 17 ■ Concorrência com futures.....	560
Exemplo: downloads da Web em três estilos	560
Um script para download sequencial	563
Fazendo download com <code>concurrent.futures</code>	565
Onde estão os futures?	566
E/S bloqueante e a GIL.....	570
Iniciando processos com <code>concurrent.futures</code>	571
Fazendo experimentos com <code>Executor.map</code>	573
Downloads com exibição de progresso e tratamento de erros	576
Tratamento de erros nos exemplos da série <code>flags2</code>	581
Usando <code>futures.as_completed</code>	584
Alternativas com <code>threading</code> e <code>multiprocessing</code>	587
Resumo do capítulo	587
Leituras complementares.....	588
Capítulo 18 ■ Concorrência com <code>asyncio</code>	594
Thread versus corrotina: uma comparação	596
<code>asyncio.Future</code> : não bloqueante por design	603
<code>yield from</code> com futures, tasks e corrotinas.....	604
Fazendo download com <code>asyncio</code> e <code>aiohttp</code>	605
Dando voltas em chamadas bloqueantes	610
Melhorando o script para download com <code>asyncio</code>	612
Usando <code>asyncio.as_completed</code>	613
Usando um executor para evitar bloqueio do loop de eventos	619
De callbacks a futures e corrotinas	620
Fazendo várias requisições para cada download	623
Escrevendo servidores com <code>asyncio</code>	626
Um servidor TCP com <code>asyncio</code>	627
Um servidor web com <code>aiohttp</code>	632
Clientes mais inteligentes para melhorar a concorrência	636
Resumo do capítulo	637
Leituras complementares.....	638

Parte VI ■ Metaprogramação 643

Capítulo 19 ■ Atributos dinâmicos e propriedades 644

Processando dados com atributos dinâmicos 645

 Explorando dados JSON ou similares com atributos dinâmicos..... 647

 O problema do nome de atributo inválido 651

 Criação flexível de objetos com `__new__`..... 652

 Reestruturando o feed da OSCON com `shelve` 654

 Recuperação de registros relacionados usando propriedades..... 658

Usando uma propriedade para validação de atributo..... 665

 LineItem tomada #1: classe para um item de um pedido 665

 LineItem tomada #2: uma propriedade com validação 666

Uma visão apropriada das propriedades..... 667

 Propriedades encobrem atributos de instância..... 669

 Documentação de propriedades..... 672

Implementando uma fábrica de propriedades..... 673

Tratando a remoção de atributos 676

Atributos e funções essenciais para tratamento de atributos 677

 Atributos especiais que afetam o tratamento de atributos 677

 Funções embutidas para tratamento de atributos 678

 Métodos especiais para tratamento de atributos 679

Resumo do capítulo 681

Leituras complementares..... 681

Capítulo 20 ■ Descritores de atributos 687

Exemplo de descritor: validação de atributos 687

 LineItem tomada #3: um descritor simples 688

 LineItem tomada #4: nomes automáticos para atributos de armazenagem . 693

 LineItem tomada #5: um novo tipo descritor 700

Descritores dominantes e não dominantes 703

 Descritor dominante 705

 Descritor dominante sem `__get__` 706

 Descritor não dominante 707

 Sobrescrevendo um descritor na classe..... 709

Métodos são descritores 709

Dicas para uso de descritores 712

Docstring de descritores e controle de remoção..... 714

Resumo do capítulo 715

Leituras complementares..... 716

Capítulo 21 ■ Metaprogramação com classes.....	718
Uma fábrica de classes	719
Um decorador de classe para personalizar descritores	722
O que acontece quando: tempo de importação versus tempo de execução.....	725
Exercícios dos instantes de avaliação	726
Básico sobre metaclasses	730
Exercício do instante de avaliação de metaclasses.....	732
Uma metaclasses para personalizar descritores.....	736
Método especial __prepare__ de metaclasses.....	738
Classes como objetos	741
Resumo do capítulo	742
Leituras complementares.....	743
Posfácio	747
Leituras complementares.....	749
Apêndice A ■ Scripts auxiliares.....	751
Capítulo 3: teste de desempenho do operador in	751
Capítulo 3: comparar padrões de bits de hashes	754
Capítulo 9: uso de RAM com e sem __slots__.....	754
Capítulo 14: script isis2json.py para conversão de banco de dados.....	755
Capítulo 16: simulação de eventos discretos para a frota de táxis.....	761
Capítulo 17: Exemplos com criptografia.....	766
Capítulo 17: exemplos de cliente HTTP para flags2.....	769
Capítulo 19: Scripts e testes para agenda da OSCON	775
Jargão de Python	781
Sobre o autor	798
Colofão	799

Eis o plano: quando uma pessoa usar uma funcionalidade que você não entende, simplesmente atire nela. Isso é mais fácil que aprender algo novo e, em pouco tempo, os únicos programadores vivos estarão codando em um minúsculo subconjunto facilmente compreensível do Python 0.9.6 <piscadela>.^{1 2}

— Tim Peters

Core developer lendário e autor de The Zen of Python

“Python é uma linguagem de programação poderosa e fácil de aprender.” Essas são as primeiras palavras do *Python Tutorial* oficial (Tutorial do Python, <https://docs.python.org/3/tutorial/>). Isso é verdade, mas há um porém: como a linguagem é fácil de aprender e de ser colocada em uso, muitos programadores que usam Python aproveitam somente uma fração de seus recursos mais poderosos.

Um desenvolvedor experiente pode começar a escrever um programa Python útil em questão de horas. À medida que as primeiras horas produtivas se transformam em semanas e em meses, muitos programadores continuam escrevendo código Python com um sotaque bem marcado, proveniente de linguagens que aprenderam anteriormente. Mesmo que Python seja a primeira linguagem que você tenha aprendido, muitas vezes no ambiente acadêmico e em livros introdutórios, ela é apresentada evitando-se cuidadosamente os recursos específicos da linguagem.

Como professor que apresenta Python a programadores com experiência em outras linguagens, vejo outro problema que este livro procura abordar: só sentimos falta daquilo que conhecemos. Qualquer pessoa que conheça outra linguagem poderá supor que o Python trate expressões regulares e procurará esse recurso em sua documentação. No entanto, se você jamais viu desempacotamento de tuplas e descritores antes, provavelmente você não os procurará, e poderá acabar não utilizando esses recursos simplesmente porque eles são específicos do Python.

1 N.T.: Essa e todas as demais citações deste livro foram traduzidas livremente de acordo com a citação original em inglês.

2 Mensagem para o grupo Usenet comp.lang.python em 23 de dezembro de 2002: “Acrimony in c.l.p.” (<https://mail.python.org/pipermail/python-list/2002-December/>).

Este livro não é uma referência exaustiva de A a Z para Python. A ênfase da obra está nos recursos da linguagem exclusivos do Python ou que não se encontram em muitas outras linguagens populares. Este também, em sua maior parte, é um livro sobre o núcleo da linguagem e algumas de suas bibliotecas. Raramente falarei sobre os pacotes que não estejam na biblioteca-padrão, apesar de o índice de pacotes do Python (PyPI) atualmente listar mais de 60 mil bibliotecas e muitas delas serem extremamente úteis.

A quem este livro se destina

Este livro foi escrito para programadores que usam Python e que queiram se tornar proficientes em Python 3. Se você conhece Python 2, porém está disposto a migrar para o Python 3.4 ou para uma versão mais recente, não deverá ter problemas. Quando escrevi este livro, a maioria dos programadores Python profissionais usava Python 2, por isso tomei o cuidado de destacar os recursos do Python 3 que possam ser novidade para esse público-alvo.

No entanto *Python fluente* trata de como tirar o máximo de proveito de Python 3.4, e não descreverei as correções necessárias para fazer o código funcionar em versões mais antigas. A maior parte dos exemplos deverá rodar em Python 2.7 com pouca ou nenhuma alteração, mas, em alguns casos, fazer o porte para uma versão mais antiga exigirá uma reescrita significativa.

Apesar do que foi dito, acredito que este livro poderá ser útil mesmo para quem precisa continuar usando Python 2.7, pois os conceitos essenciais permanecem iguais. Python 3 não é uma linguagem nova, e a maior parte das diferenças pode ser aprendida em uma tarde. *What's New in Python 3.0* (O que há de novo no Python 3, <https://docs.python.org/3.0/whatsnew/3.0.html>) é um bom ponto de partida. É claro que houve mudanças desde que Python 3.0 foi lançado em 2009, porém nenhuma tão importante quanto aquelas da versão 3.0.

Se você não tem certeza de que conhece Python o suficiente para acompanhar o livro, revise os tópicos do *Python Tutorial* oficial (Tutorial do Python, <https://docs.python.org/3/tutorial/>). Os assuntos discutidos no tutorial não serão explicados neste livro, exceto algumas funcionalidades que são novidade no Python 3.

A quem este livro não se destina

Se você está começando a aprender Python agora, será difícil acompanhar este livro. Não é só isso; se você lê-lo cedo demais em sua jornada por Python, o livro poderá passar a impressão de que todo script Python precisa usar métodos especiais e truques de metaprogramação. Abstração prematura é tão ruim quanto otimização prematura.

Como este livro está organizado

O público-alvo principal deste livro não deverá ter problemas em pular diretamente para qualquer capítulo. Mas cada uma das seis partes forma um livro dentro do livro. Organizei os capítulos de cada parte para serem lidos em sequência.

Procurei enfatizar o que já está disponível antes de discutir a criação de seus próprios objetos. Por exemplo, na Parte II, o capítulo 2 discute os tipos de sequência que já estão prontos para serem usados, incluindo alguns que não recebem muita atenção, como `collections.deque`. Criar sequências definidas pelo usuário é um assunto tratado somente na Parte IV, em que também veremos como aproveitar as ABCs (abstract base classes, ou classes-base abstratas) de `collections.abc`. A criação de suas próprias ABCs será discutida posteriormente na Parte IV, pois acredito que é importante estar à vontade para usar uma ABC antes de criar a sua própria classe desse tipo.

Essa abordagem tem algumas vantagens. Para começar, saber o que já está pronto para ser usado evita que você reinvente a roda. Usamos as classes de coleções existentes com muito mais frequência do que implementamos nossas próprias coleções, e podemos dar mais atenção aos usos avançados das ferramentas disponíveis se adiarmos a discussão sobre a criação de novas classes. Também é mais provável herdar de ABCs existentes do que criar uma nova ABC do zero. Por fim, acredito que é mais fácil entender as abstrações depois de vê-las em ação.

A desvantagem dessa estratégia são as referências antecipadas espalhadas pelos capítulos. Espero que isso seja mais fácil de tolerar, agora que você sabe por que decidi seguir esse caminho.

Eis os principais assuntos discutidos em cada parte do livro:

Parte I

Um único capítulo sobre o modelo de dados do Python que explica como os métodos especiais (por exemplo, `__repr__`) são o segredo para o comportamento consistente dos objetos de todos os tipos – em uma linguagem que é admirada pela sua consistência. Compreender as diversas facetas do modelo de dados é o assunto da maior parte do restante do livro, porém o capítulo 1 dá uma visão geral.

Parte II

Os capítulos dessa parte discutem o uso das coleções – sequências, mapeamentos e conjuntos, assim como a divisão entre `str` e `bytes` –, motivo de muita comemoração entre os usuários do Python 3 e de muito sofrimento para os usuários do Python 2 que ainda não fizeram a migração de suas bases de código. Os principais objetivos são recordar o que já está disponível e explicar alguns comportamentos que, às vezes, causam surpresa, por exemplo, a reordenação das chaves de `dict` quando

não estamos olhando ou as complicações relacionadas à ordenação de strings Unicode dependentes de localidade. Para atingir esses objetivos, a discussão às vezes será genérica e ampla (por exemplo, quando muitas variações de sequências e de mapeamentos forem apresentadas), outras vezes será profunda (por exemplo, ao mergulharmos de cabeça nas tabelas hash subjacentes aos tipos `dict` e `set`).

Parte III

Nessa parte, discutiremos as funções como objetos de primeira classe na linguagem: o que isso quer dizer, como elas afetam alguns padrões de projeto populares e como implementar decoradores de funções usando closures. Além disso, discutiremos o conceito geral de “callables” (objetos invocáveis) em Python, os atributos de função, a introspecção, as anotações em parâmetros e a nova declaração `nonlocal` de Python 3.

Parte IV

Nessa parte, o foco está na criação de classes. Na Parte II, a declaração `class` aparece em poucos exemplos; a Parte IV apresenta muitas classes. Como qualquer linguagem orientada a objetos (OO), Python tem seu conjunto particular de funcionalidades que podem ou não estar presentes na linguagem em que você e eu aprendemos a programar com classes. Os capítulos explicam como funcionam as referências, o que realmente significa mutabilidade, o ciclo de vida das instâncias, como criar suas próprias coleções e ABCs, como lidar com herança múltipla e como implementar a sobrecarga de operadores – quando isso fizer sentido.

Parte V

Nessa parte, serão discutidas as construções da linguagem e as bibliotecas que vão além do fluxo de controle sequencial com condicionais, loops e sub-rotinas. Começamos com os geradores (generators); em seguida, veremos os gerenciadores de contexto (context managers) e as corrotinas (coroutines), incluindo a nova sintaxe `yield from` – difícil de compreender, porém poderosa. A Parte V encerra com uma introdução geral à concorrência moderna em Python com `collections`, `futures` (usando `threads` e processos internamente com a ajuda de `futures`) e mostra como realizar I/O orientado a eventos com `asyncio` (tirando proveito das `futures` com corrotinas e `yield from`).

Parte VI

Essa parte começa com uma revisão das técnicas para a construção de classes com atributos criados dinamicamente a fim de lidar com dados semiestruturados, por exemplo, com massas de dados JSON. Em seguida, discutiremos o sistema de propriedades antes de mergulhar no modo como o acesso aos atributos de um

objeto funciona no nível mais baixo em Python usando descritores (descriptors). O relacionamento entre funções, métodos e descritores será explicado. Ao longo da Parte VI, a implementação passo a passo de uma biblioteca de validação de campos revelará problemas sutis que levarão ao uso de ferramentas sofisticadas no último capítulo: os decoradores de classe e as metaclasses.

Uma abordagem “mão na massa”

Com frequência, usaremos o console interativo do Python para explorar a linguagem e as bibliotecas. Acho importante enfatizar o poder dessa ferramenta de aprendizado, particularmente para os leitores que tiveram mais experiência com linguagens estáticas e compiladas que não ofereçam um read-eval-print loop (REPL).

Um dos pacotes-padrões de testes para Python, o doctest (<https://docs.python.org/3/library/doctest.html>), funciona simulando sessões de console e verificando se as expressões produzem os resultados esperados. Usei o doctest para conferir a maior parte do código deste livro, incluindo as listagens do console. Você não precisa usar nem sequer conhecer o doctest para acompanhar o livro; a principal característica dos doctests está no fato de parecerem transcrições de sessões interativas de console do Python, portanto você poderá testar facilmente as demonstrações por conta própria.

Às vezes, explicarei o que queremos fazer mostrando um doctest antes de apresentar o código que o fará passar. Definir claramente o que deve ser feito antes de pensar em como fazê-lo ajuda a focar nossos esforços de programação. Escrever os testes antes é a base do TDD (Test Driven Development, ou Desenvolvimento orientado a testes), e também acredito que isso ajuda no ensino. Se você não tiver familiaridade com o doctest, dê uma olhada em sua documentação (<https://docs.python.org/3/library/doctest.html>) e no repositório de códigos-fonte deste livro (<https://github.com/fluentpython/example-code>). Você verá que será possível verificar se a maior parte do código do livro está correta digitando `python3 -m doctest example_script.py` no terminal de seu sistema operacional.

Hardware usado para medições de tempo

Este livro tem alguns benchmarks e medições simples de tempo. Esses testes foram realizados em um dos laptops que usei para escrever: um MacBook Pro 13” de 2011 com uma CPU Intel Core i7 de 2.7 GHz, 8 GB de RAM e um disco rígido convencional, e um MacBook Air 13” de 2014 com uma CPU Intel Core i5 de 1.4 GHz, 4 GB de RAM e um disco de estado sólido. O MacBook Air tem uma CPU mais lenta e menos RAM, porém sua RAM é mais rápida (1.600 *versus* 1.333 MHz) e o SSD é muito mais rápido que o HD. No uso cotidiano, não consigo dizer qual dos dois é mais rápido.

Meu ponto de vista

Uso, ensino e discuto Python desde 1998, e gosto de estudar e de comparar as linguagens de programação, seu design e a teoria por trás delas. No final dos capítulos, coloquei minha visão pessoal sobre Python e outras linguagens em caixas intituladas “Ponto de vista”. Sinta-se à vontade para ignorá-las se não estiver interessado nessas discussões. O conteúdo delas é totalmente opcional.

Jargão do Python

Gostaria que este livro fosse não só sobre Python, mas também sobre a cultura ao redor da linguagem. Durante mais de vinte anos de comunicações, a comunidade Python desenvolveu o seu próprio jargão e suas siglas. No final do livro, a seção Jargão do Python contém uma lista de termos que têm significados especiais entre os Pythonistas.

Versão do Python discutida

Testei todo o código deste livro em Python 3.4 – isto é, CPython 3.4, que é a implementação mais popular de Python, escrita em C. Há apenas uma exceção: “O novo operador infix `@` no Python 3.5” na página 432 mostra o operador `@`, que é aceito somente pelo Python 3.5.

Quase todo o código do livro deverá funcionar com qualquer interpretador compatível com Python 3.x, incluindo o PyPy3 2.4.0, que é compatível com o Python 3.2.5. As exceções dignas de nota são os exemplos que usam `yield from` e `asyncio`, disponíveis somente no Python 3.3 ou em versões mais recentes.

A maior parte do código também deverá funcionar no Python 2.7 com pequenas alterações, exceto os exemplos relacionados com Unicode no capítulo 4 e as exceções já apontadas para as versões de Python 3 anteriores à versão 3.3.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

Observe que, quando uma quebra de linha ocorrer no meio de um termo com `largura_constante`, um hífen não será acrescentado – ele poderia ser confundido como parte do termo.

Largura constante em negrito

Mostra comandos ou outro texto que devam ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.



Esse elemento significa uma dica ou sugestão.



Esse elemento significa uma nota geral.



Esse elemento indica um aviso ou cuidado.

Uso dos exemplos de código

Todos os scripts e a maior parte dos trechos de código que aparecem no livro estão disponíveis no repositório de códigos de *Python fluente* no GitHub (<https://github.com/fluentpython/example-code>).

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*Fluent Python* de Luciano Ramalho (O’Reilly). Copyright 2015 Luciano Ramalho, 978-1-491-94600-8.”

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: novatec@novatec.com.br.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

<http://www.novatec.com.br/catalogo/7522462-pythonfluente>

- Página da edição original em inglês

<http://bit.ly/fluente-python>

Para obter mais informações sobre os livros da Novatec, acesse nosso site em <http://www.novatec.com.br>.

Agradecimentos

O conjunto de peças de xadrez Bauhaus de Josef Hartwig é um exemplo de design excelente: bonito, simples e claro. Guido von Rossum, filho de um arquiteto e irmão de um designer de fontes premiado, criou uma obra-prima do design de linguagens. Adoro ensinar Python porque ela é bonita, simples e clara.

Alex Martelli e Anna Ravenscroft foram as primeiras pessoas que viram o esqueleto deste livro e me incentivaram a submeter o projeto à O'Reilly para publicação. Seus livros me ensinaram Python idiomático e são modelos de clareza, precisão e de profundidade na escrita técnica. Os mais de cinco mil posts de Alex no Stack Overflow (<http://stackoverflow.com/users/95810/alex-martelli>) são uma fonte de insights sobre a linguagem e o seu uso apropriado.

Martelli e Ravenscroft também foram revisores técnicos deste livro, juntamente com Lennart Regebro e Leonardo Rochaël. Todos que fizeram parte dessa equipe maravilhosa de revisores técnicos têm pelo menos quinze anos de experiência com Python e fizeram muitas contribuições a projetos Python de grande impacto em contato muito próximo com outros desenvolvedores da comunidade. Juntos, eles me enviaram centenas de correções, sugestões, perguntas e opiniões, o que adicionou um tremendo valor ao livro. Victor Stinner gentilmente revisou o capítulo 18, trazendo o seu expertise como mantenedor do `asyncio` à equipe de revisão técnica. Foi um grande privilégio e um prazer trabalhar com eles nesses últimos meses.

A editora Meghan Blanchette foi uma excelente mentora, ajudando-me a melhorar a organização e o fluxo do livro, dizendo-me quando ele estava maçante e evitando

que eu atrasasse mais ainda. Brian MacDonald editou os capítulos da Parte III enquanto Meghan estava ausente. Gostei de trabalhar com eles e com todos com quem tive contato na O'Reilly, incluindo a equipe de desenvolvimento e suporte ao Atlas (Atlas é a plataforma de publicação de livros da O'Reilly, que tive o prazer de usar para escrever este livro).

Mario Domenech Goulart enviou inúmeras sugestões detalhadas desde a primeira versão *Early Release*. Também recebi feedbacks valiosos de Dave Pawson, Elias Dorneles, Leonardo Alexandre Ferreira Leite, Bruce Eckel, J. S. Bueno, Rafael Gonçalves, Alex Chiaranda, Guto Maia, Lucas Vido e Lucas Brunialti.

Ao longo dos anos, muitas pessoas me incentivaram a ser autor, porém as mais persuasivas foram Rubens Prates, Aurelio Jargas, Rudá Moura e Rubens Altimari. Mauricio Bussab abriu muitas portas para mim, incluindo minha primeira tentativa real de escrever um livro. Renzo Nuccitelli apoiou este projeto o tempo todo, mesmo quando isso significou um início mais lento à nossa parceria em *python.pro.br*.

A maravilhosa comunidade Python brasileira é bem-informada, generosa e divertida. O grupo Python Brasil (<https://groups.google.com/group/python-brasil>) é composto de milhares de pessoas, e nossas conferências nacionais reúnem centenas delas, porém as mais influentes em minha jornada como Pythonista foram Leonardo Rochaël, Adriano Petrich, Daniel Vainsencher, Rodrigo RBP Pimentel, Bruno Gola, Leonardo Santagada, Jean Ferri, Rodrigo Senra, J. S. Bueno, David Kwast, Luiz Irber, Osvaldo Santana, Fernando Masanori, Henrique Bastos, Gustavo Niemayer, Pedro Werneck, Gustavo Barbieri, Lalo Martins, Danilo Bellini e Pedro Kroger.

Dorneles Tremea foi um ótimo amigo (extremamente generoso com seu tempo e seu conhecimento), um hacker incrível e o líder mais inspirador da Associação Python Brasil. Ele nos deixou cedo demais.

Meus alunos ao longo dos anos me ensinaram muito por meio de suas perguntas, seus insights, feedbacks e suas soluções criativas para os problemas. Érico Andrei e a Simples Consultoria me possibilitaram focar em ser professor de Python pela primeira vez.

Martijn Faassen foi meu mentor de Grok e compartilhou insights de valor inestimável comigo sobre Python e homens de Neanderthal. Seu trabalho e o de Paul Everitt, Chris McDonough, Tres Seaver, Jim Fulton, Shane Hathaway, Lennart Regebro, Alan Runyan, Alexander Limi, Martijn Pieters, Godefroid Chapelle e outros dos planetas Zope, Plone e Pyramid foram decisivos em minha carreira. Graças ao Zope e por surfar a primeira onda da web, pude começar a ganhar a vida com o Python em 1998. José Octavio Castro Neves foi meu sócio na primeira empresa de software especializada em Python no Brasil.

Tenho muitos gurus na comunidade Python internacional para que possa listar todos eles, porém, além daqueles que já mencionei, devo muito a Steve Holden, Raymond Hettinger, A. M. Kuchling, David Beazley, Fredrik Lundh, Doug Hellmann, Nick Coghlan, Mark Pilgrim, Martijn Pieters, Bruce Eckel, Michele Simionato, Wesley Chun, Brandon Craig Rhodes, Philip Guo, Daniel Greenfeld, Audrey Roy e Brett Slatkin por me mostrarem maneiras novas e melhores de ensinar Python.

A maioria destas páginas foi escrita em meu home office e em dois laboratórios: o CoffeeLab e o Garoa Hacker Clube. O CoffeeLab (<http://coffeelab.com.br/>) é a sede dos geeks aficionados por cafeína na Vila Madalena em São Paulo, no Brasil. O Garoa Hacker Clube (<https://garoa.net.br/>) é um hackerspace aberto a todos: um laboratório comunitário onde qualquer pessoa pode experimentar novas ideias livremente.

A comunidade Garoa proporcionou inspiração, infraestrutura e *slack*. Acho que Aleph gostaria deste livro.

Minha mãe, Maria Lucia, e meu pai, Jairo, sempre me apoiaram em todos os aspectos. Gostaria que ele estivesse aqui para ver o livro; fico feliz em poder compartilhá-lo com ela.

Minha esposa, Marta Mello, aturou um marido que esteve sempre trabalhando durante quinze meses, porém continuou me apoiando e me encorajou em alguns momentos críticos do projeto em que cogitei desistir da maratona.

Obrigado a todos vocês por tudo.

PARTE I

Prólogo

CAPÍTULO 1

Modelo de dados do Python

O senso estético de Guido no design da linguagem é incrível. Conheci muitos bons designers de linguagem que poderiam ter criado belas linguagens em teoria, mas que ninguém jamais usaria, porém Guido é uma dessas raras pessoas capazes de criar uma linguagem apenas um pouco menos bela teoricamente, mas com a qual é um prazer escrever programas.¹

— Jim Hugunin

Criador do Jython, cocriador do AspectJ, arquiteto do DLR do .Net

Uma das melhores qualidades de Python é a sua consistência. Depois de trabalhar com Python por um tempo, você começará a ter palpites corretos e bem fundamentados sobre recursos que você ainda nem estudou na documentação.

No entanto, se você aprendeu outra linguagem orientada a objetos antes de Python, poderá achar estranho escrever `len(collection)` em vez de `collection.len()`. Essa aparente estranheza é a ponta de um iceberg que, quando bem compreendido, é o segredo de tudo que consideramos *pythônico*. O iceberg é chamado de Python data model (modelo de dados de Python) e descreve a API que você pode usar para fazer com que seus próprios objetos interajam bem com os recursos mais idiomáticos da linguagem.

Podemos pensar no modelo de dados como uma descrição de Python na forma de um framework. Ele formaliza as interfaces dos blocos de construção da própria linguagem, por exemplo, as sequências, os iteradores, as funções, as classes, os gerenciadores de contexto e assim por diante.

Ao codificar com qualquer framework, você investe bastante tempo implementando métodos que são chamados pelo framework. O mesmo acontece quando você aproveita o modelo de dados do Python. O interpretador Python chama métodos especiais para realizar operações básicas em objetos, geralmente acionados por uma sintaxe especial. Os nomes dos métodos especiais são sempre escritos com underscores duplos

¹ História do Jython (http://hugunin.net/story_of_jython.html), escrita como prefácio do livro *Jython Essentials* (<http://bit.ly/jython-essentials>) (O'Reilly, 2002), por Samuele Pedroni e Noel Rappin.

no início e no fim (ou seja, `__getitem__`). Por exemplo, a sintaxe `obj[key]` é tratada pelo método especial `__getitem__`. Para avaliar `my_collection[key]`, o interpretador chama `my_collection.__getitem__(key)`.

Os nomes dos métodos especiais permitem que seus objetos implementem, tratem e interajam com as construções básicas da linguagem, tais como:

- Iteração
- Coleções
- Acesso a atributos
- Sobrecarga de operadores
- Chamada de funções e de métodos
- Criação e destruição de objetos
- Representação e formatação de strings
- Contextos gerenciados (ou seja, blocos `with`)



“mágico” e “dunder”

O termo *método mágico* é uma gíria para método especial, porém, quando falamos de um método específico como `__getitem__`, alguns desenvolvedores Python simplesmente dizem “under-under-getitem”, que é ambíguo, pois a sintaxe `__x` tem outro significado especial ^(*). Ser exato e pronunciar “under-under-getitem-under-under” é cansativo, portanto sigo a dica do autor e professor Steve Holden e digo “dunder-getitem”. Todos os pythonistas experientes entendem essa forma abreviada. Por isso os métodos especiais também são conhecidos como *métodos dunder* (dunder methods). ^(**)

(*) Veja a seção “Atributos privados e “protegidos” em Python” na página 304.

(**) Ouvi pessoalmente o termo “dunder” pela primeira vez de Steve Holden. A Wikipedia atribui (<http://bit.ly/1Vm72Mf>) os primeiros registros escritos do termo “dunder” a Mark Johnson e a Tim Hochberg em respostas à pergunta “How do you pronounce __ (double underscore)?” (Como você pronuncia __ (underscores duplos?)) na python-list em 26 de setembro de 2002: mensagem de Johnson (<https://mail.python.org/pipermail/python-list/2002-September/>); mensagem de Hochberg (<https://mail.python.org/pipermail/python-list/2002-September/>) – onze minutos depois.

Um baralho pythônico

A seguir, apresentamos um exemplo bem simples, mas que mostra a eficácia da implementação de apenas dois métodos especiais, `__getitem__` e `__len__`.

O exemplo 1.1 mostra uma classe que representa um conjunto de cartas de baralho.

Exemplo 1.1 – Um baralho como uma sequência de cartas

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

O primeiro aspecto a ser observado é o uso de `collections.namedtuple` para criar uma classe simples que representa as cartas individuais. Desde Python 2.6, `namedtuple` pode ser usada para criar classes de objetos que sejam apenas grupos de atributos, sem métodos próprios, como um registro de banco de dados. No exemplo, ela foi usada para fornecer uma bela representação das cartas do baralho, conforme mostrado nesta sessão do console:

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

No entanto o ponto principal desse exemplo é a classe `FrenchDeck`. Ela é concisa, porém bastante funcional. Para começar, como qualquer coleção Python padrão, o baralho responde à função `len()`, devolvendo o número de cartas que ele contém:

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

Ler cartas específicas do baralho – por exemplo, a primeira ou a última – é simples; basta usar `deck[0]` ou `deck[-1]`, e é isso o que o método `__getitem__` faz acontecer:

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

Devemos criar um método para selecionar uma carta aleatória? Não é necessário. O Python já tem uma função para obter um item aleatório de uma sequência: `random.choice`. Podemos simplesmente usá-la em uma instância do baralho:

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

Acabamos de ver duas vantagens de usar os métodos especiais para tirar proveito do modelo de dados do Python:

- Os usuários de suas classes não precisarão memorizar nomes arbitrários de métodos para realizar operações comuns (“Como obter a quantidade de itens? Devo usar `.size()`, `.length()` ou o quê?”).
- É mais fácil se beneficiar da rica biblioteca-padrão do Python do que reinventar a roda, como no caso da função `random.choice`.

Mas o que já temos é ainda melhor.

Como nosso `__getitem__` delega a responsabilidade ao operador `[]` de `self._cards`, nosso baralho já suporta fatiamento (slicing). Veja como podemos inspecionar as três primeiras cartas de um baralho novo e, em seguida, escolher somente os ases, começando no índice 12 e avançando treze cartas de cada vez:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Só por implementar o método especial `__getitem__`, nosso baralho também é iterável:

```
>>> for card in deck: # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

A iteração também pode ser feita no baralho na ordem inversa:

```
>>> for card in reversed(deck): # doctest: +ELLIPSIS
...     print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```



Reticências em doctests

Sempre que possível, as listagens do console do Python neste livro foram extraídas de doctests para garantir a exatidão. Quando a saída for longa demais, a parte omitida será marcada com reticências (...), como na última linha do código anterior. Em casos como esse, usamos a diretiva `#doctest: +ELLIPSIS` para fazer o doctest passar. Se estiver testando esses exemplos no console interativo, você poderá omitir totalmente as diretivas do doctest.

A iteração muitas vezes é implícita. Se uma coleção não tiver nenhum método `__contains__`, o operador `in` fará uma varredura sequencial. Em nosso caso em particular, `in` funciona com a classe `FrenchDeck` porque ela é iterável. Veja só:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

E o que dizer da ordenação? Um sistema comum de classificação de cartas é pelo valor (com os ases sendo as cartas mais altas) e, em seguida, pelo naipe na seguinte ordem: espadas (o maior), depois copas, ouros e paus (o menor). Eis uma função que classifica as cartas de acordo com essa regra, retornando 0 para o 2 de paus e 51 para o ás de espadas:

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

Após ter definido `spades_high`, podemos agora listar o nosso baralho em ordem crescente de classificação:

```
>>> for card in sorted(deck, key=spades_high): # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
```

```
... (46 cards omitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Embora FrenchDeck herde implicitamente de `object`², sua funcionalidade mais importante não é herdada, porém resulta de aproveitarmos o modelo de dados e composição. Ao implementar os métodos especiais `__len__` e `__getitem__`, nosso FrenchDeck se comporta como uma sequência Python padrão, permitindo que se beneficie de recursos essenciais da linguagem (por exemplo, iteração e fatiamento) e da biblioteca-padrão, conforme mostrado nos exemplos que usaram `random.choice`, `reversed` e `sorted`. Graças à composição, as implementações de `__len__` e de `__getitem__` podem passar todo o trabalho para um objeto `list`, ou seja, `self._cards`.



E o que dizer de embaralhar?

Conforme implementado até agora, um FrenchDeck não pode ser embaralhado, pois ele é *imutável*: as cartas e suas posições não poderão ser alteradas pelos clientes, exceto se violarem o encapsulamento e manipularem o atributo `_cards` diretamente. No capítulo 11, isso será corrigido por meio da adição de um método `__setitem__` de uma linha.

Como os métodos especiais são usados

A primeira coisa a saber sobre os métodos especiais é que eles foram criados para serem chamados pelo interpretador Python, e não por você. Não escrevemos `my_object.__len__()`. Escrevemos `len(my_object)` e, se `my_object` for uma instância de uma classe definida por você, Python chamará o método de instância `__len__` que você implementou.

Porém, para os tipos embutidos (built-in) como `list`, `str`, `bytearray` e outros, o interpretador usará um atalho: a implementação de `len()` do CPython, na verdade, retorna o valor do campo `ob_size` da struct C `PyVarObject` que representa qualquer objeto embutido de tamanho variável na memória. Isso é muito mais rápido que chamar um método.

Na maioria das vezes, a chamada ao método especial será implícita. Por exemplo, a instrução `for i in x:` provoca a chamada de `iter(x)`, que, por sua vez, poderá chamar `x.__iter__()` se ele estiver disponível.

Normalmente, seu código não deverá ter muitas chamadas diretas aos métodos especiais. A menos que você esteja usando bastante metaprogramação, será mais comum

2 Em Python 2, você deverá ser explícito e escrever `FrenchDeck(object)`, porém esse é o default em Python 3.

você implementar métodos especiais do que invocá-los diretamente. O único método especial chamado frequentemente de forma direta pelo usuário é `__init__`, para invocar o inicializador da superclasse quando você implementa seu próprio `__init__`.

Se for preciso chamar um método especial, normalmente é melhor chamar a função embutida relacionada (por exemplo, `len`, `iter`, `str` etc.). Essas funções embutidas invocam o método especial correspondente, porém, com frequência, oferecem outros serviços e – para os tipos embutidos – são mais rápidas que as chamadas de métodos. Por exemplo, dê uma olhada na seção “Uma visão mais detalhada da função `iter`” na página 488 do capítulo 14.

Evite criar atributos arbitrários personalizados com a sintaxe `__foo__`, pois esses nomes poderão adquirir significados especiais no futuro, mesmo se não estiverem em uso atualmente.

Emulando tipos numéricos

Diversos métodos especiais permitem que os objetos do usuário respondam a operadores, como `+`. Veremos esse assunto com mais detalhes no capítulo 13, porém nosso objetivo aqui é ilustrar melhor o uso dos métodos especiais por meio de outro exemplo simples.

Implementaremos uma classe para representar vetores bidimensionais – ou seja, vetores euclidianos como aqueles usados em matemática e em física (veja a figura 1.1).

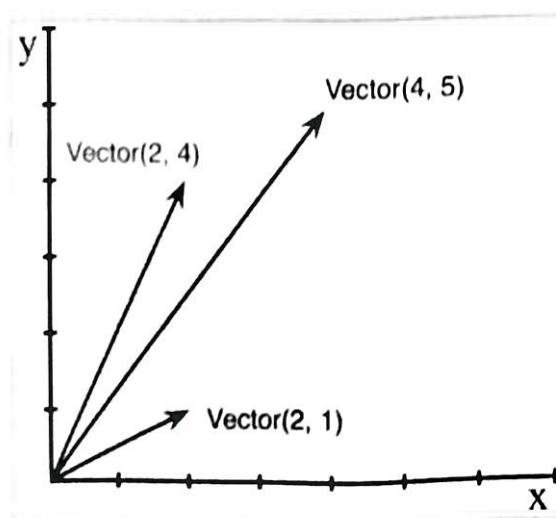


Figura 1.1 – Exemplo de soma de vetores bidimensionais; $\text{Vector}(2, 4) + \text{Vector}(2, 1)$ resulta em $\text{Vector}(4, 5)$.



O tipo embutido `complex` pode ser usado para representar vetores bidimensionais, porém nossa classe pode ser estendida para representar vetores n -dimensionais. Faremos isso no capítulo 14.

Começaremos pelo design da API de uma classe desse tipo escrevendo uma sessão simulada de console que poderá ser usada posteriormente como um doctest. O trecho de código a seguir testa a soma de vetores representada na figura 1.1:

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Observe como o operador `+` gera um resultado `Vector`, exibido de forma amigável no console.

A função embutida `abs` retorna o valor absoluto de inteiros e de números de ponto flutuante e a magnitude de números complexos (`complex`); sendo assim, para sermos consistentes, nossa API também usará `abs` para calcular a magnitude de um vetor:

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

Também podemos implementar o operador `*` para realizar uma multiplicação por escalar (ou seja, multiplicar um vetor por um número de modo a gerar um novo vetor com a mesma direção e uma magnitude multiplicada):

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

O exemplo 1.2 apresenta uma classe `Vector` que implementa as operações que acabaram de ser descritas, por meio dos métodos especiais `__repr__`, `__abs__`, `__add__` e `__mul__`.

Exemplo 1.2 – Uma classe simples de vetor bidimensional

```
from math import hypot
```

```
class Vector:
```

```
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vector(%r, %r)' % (self.x, self.y)

    def __abs__(self):
        return hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))
```

```
def __add__(self, other):  
    x = self.x + other.x  
    y = self.y + other.y  
    return Vector(x, y)  
  
def __mul__(self, scalar):  
    return Vector(self.x * scalar, self.y * scalar)
```

Observe que, apesar de termos implementado quatro métodos especiais (além de `__init__`), nenhum deles é chamado diretamente na classe ou no uso típico da classe ilustrado pelas listagens do console. Conforme mencionamos anteriormente, o interpretador Python é o único a chamar a maioria dos métodos especiais com frequência. Nas seções a seguir, discutiremos o código de cada método especial.

Representação em string

O método especial `__repr__` é chamado pela função embutida `repr` para obtermos a representação em string do objeto para inspeção. Se `__repr__` não for implementado, as instâncias dos vetores serão exibidas no console como `<Vector object at 0x10e100070>`.

O console interativo e o debugger (depurador) chamam `repr` nos resultados das expressões avaliadas, como faz a marcação `%r` na formatação clássica com o operador `%` e o marcador de conversão `!r` na nova sintaxe de formatação de strings (*Format String Syntax*, <http://bit.ly/1Vm7gD1>) usada no método `str.format`.



Falando no operador `%` e no método `str.format`, você perceberá que uso ambos neste livro, como faz a comunidade Python em geral. Cada vez mais, prefiro o `str.format`, mais versátil, porém sei que muitos pythonistas preferem o `%`, mais simples; sendo assim, é provável que vejamos ambos em códigos-fonte Python por muito tempo.

Observe que, em nossa implementação de `__repr__`, usamos `%r` para obter a representação-padrão dos atributos a serem exibidos. Essa é uma boa prática, pois mostra a diferença fundamental entre `Vector(1, 2)` e `Vector('1', '2')` – a última não funcionará no contexto desse exemplo, pois os argumentos do construtor devem ser números, e não `str`.

A string retornada por `__repr__` não deve ser ambígua e, se possível, deverá corresponder ao código-fonte necessário para recriar o objeto sendo representado. É por isso que a representação que escolhemos se parece com a chamada ao construtor da classe (por exemplo, `Vector(3, 4)`).

Compare `__repr__` com `__str__`, que é chamada pelo construtor `str()` e usada implicitamente pela função `print`. `__str__` deve retornar uma string adequada para exibição aos usuários finais.

Se você implementar somente um desses métodos especiais, escolha `__repr__` porque, quando um `__str__` personalizado não estiver disponível, o Python chamará `__repr__` como alternativa.



“Difference between `__str__` and `__repr__` in Python” (Diferença entre `__str__` e `__repr__` em Python, <http://bit.ly/1Vm7j1N>) é uma pergunta do Stack Overflow com excelentes contribuições dos pythonistas Alex Martelli e Martijn Pieters.

Operadores aritméticos

O exemplo 1.2 implementa dois operadores, `+` e `*`, para mostrar o uso básico de `__add__` e de `__mul__`. Observe que, em ambos os casos, os métodos criam e devolvem uma nova instância de `Vector` e não modificam nenhum operando – `self` ou `other` são simplesmente lidos. Esse é o comportamento esperado dos operadores infixos: novos objetos são criados e os operandos não são alterados. Terei muito mais a dizer sobre esse assunto no capítulo 13.



Conforme implementado, o exemplo 1.2 permite multiplicar um `Vector` por um número, mas não um número por um `Vector`, o que viola a propriedade comutativa da multiplicação. Corrigiremos isso com o método especial `__rmul__` no capítulo 13.

Valor booleano de um tipo definido pelo usuário

Embora Python tenha um tipo `bool`, qualquer objeto é aceito em um contexto booleano, por exemplo, na expressão que controla uma instrução `if` ou `while` ou como operandos de `and`, `or` e `not`. Para determinar se um valor `x` é *verdadeiro* ou *falso*, Python aplica `bool(x)`, que sempre devolve `True` ou `False`.

Por padrão, as instâncias das classes definidas pelo usuário são consideradas verdadeiras, a menos que `__bool__` ou `__len__` esteja implementado. Basicamente, `bool(x)` chama `x.__bool__()` e utiliza o resultado. Se `__bool__` não estiver implementado, Python tentará chamar `x.__len__()` e, se o resultado for zero, `bool` retornará `False`. Caso contrário, `bool` retornará `True`.

Nossa implementação de `__bool__` é conceitualmente simples: ele devolverá `False` se a magnitude do vetor for zero e `True`, caso contrário. Convertamos a magnitude em um valor booleano usando `bool(abs(self))`, pois espera-se que `__bool__` devolva um booleano.

Note como o método especial `__bool__` permite que seus objetos sejam consistentes com as regras de testes para contextos booleanos definidas no capítulo “Built-in Types” (Tipos embutidos, <https://docs.python.org/3/library/stdtypes.html#truth>) da documentação da biblioteca-padrão de Python (*The Python Standard Library*).



Uma implementação mais rápida de `Vector.__bool__` é:

```
def __bool__(self):
    return bool(self.x or self.y)
```

É mais difícil de ler, porém evita a passagem por `abs`, `__abs__`, os quadrados e a raiz quadrada. A conversão explícita para `bool` é necessária porque `__bool__` deve retornar um booleano e `or` retorna qualquer um dos operandos como ele é: `x or y` é avaliado com `x` se ele for *verdadeiro*, caso contrário, o resultado será `y`, independentemente de seu valor.

Visão geral dos métodos especiais

O capítulo “Data Model” (Modelo de dados, <http://docs.python.org/3/reference/datamodel.html>) de *The Python Language Reference* (Guia de referência à linguagem Python) lista os nomes de 83 métodos especiais; 47 dos quais são usados para implementar operadores aritméticos, bit a bit (bitwise) e de comparação.

Para ter uma visão geral do que está disponível, veja as tabelas 1.1 e 1.2.



O agrupamento mostrado nas tabelas a seguir não é exatamente o mesmo que está na documentação oficial.

Tabela 1.1 – Nomes dos métodos especiais (não inclui operadores)

Categoria	Nomes dos métodos
Representação em string/bytes	<code>__repr__</code> , <code>__str__</code> , <code>__format__</code> , <code>__bytes__</code>
Conversão para número	<code>__abs__</code> , <code>__bool__</code> , <code>__complex__</code> , <code>__int__</code> , <code>__float__</code> , <code>__hash__</code> , <code>__index__</code>
Emulação de coleções	<code>__len__</code> , <code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__contains__</code>
Iteração	<code>__iter__</code> , <code>__reversed__</code> , <code>__next__</code>
Emulação de invocáveis	<code>__call__</code>
Gerenciamento de contexto	<code>__enter__</code> , <code>__exit__</code>
Criação e destruição de instâncias	<code>__new__</code> , <code>__init__</code> , <code>__del__</code>

Categoria	Nomes dos métodos
Gerenciamento de atributos	<code>__getattr__</code> , <code>__getattribute__</code> , <code>__setattr__</code> , <code>__delattr__</code> , <code>__dir__</code>
Descritores de atributos	<code>__get__</code> , <code>__set__</code> , <code>__delete__</code>
Serviços de classes	<code>__prepare__</code> , <code>__instancecheck__</code> , <code>__subclasscheck__</code>

Tabela 1.2 – Nomes dos métodos especiais para operadores

Categoria	Nomes dos métodos e operadores relacionados
Operadores numéricos unários	<code>__neg__</code> -, <code>__pos__</code> +, <code>__abs__</code> abs()
Operadores de comparação rica	<code>__lt__</code> <, <code>__le__</code> <=, <code>__eq__</code> ==, <code>__ne__</code> !=, <code>__gt__</code> >, <code>__ge__</code> >=
Operadores aritméticos	<code>__add__</code> +, <code>__sub__</code> -, <code>__mul__</code> *, <code>__truediv__</code> /, <code>__floordiv__</code> //, <code>__mod__</code> %, <code>__divmod__</code> divmod(), <code>__pow__</code> ** ou pow(), <code>__round__</code> round()
Operadores aritméticos reversos	<code>__radd__</code> , <code>__rsub__</code> , <code>__rmul__</code> , <code>__rtruediv__</code> , <code>__rfloordiv__</code> , <code>__rmod__</code> , <code>__rdivmod__</code> , <code>__rpow__</code>
Operadores aritméticos de atribuição combinada	<code>__iadd__</code> , <code>__isub__</code> , <code>__imul__</code> , <code>__itruediv__</code> , <code>__ifloordiv__</code> , <code>__imod__</code> , <code>__ipow__</code>
Operadores bit a bit (bitwise)	<code>__invert__</code> ~, <code>__lshift__</code> <<, <code>__rshift__</code> >>, <code>__and__</code> &, <code>__or__</code> , <code>__xor__</code> ^
Operadores bit a bit reversos	<code>__rlshift__</code> , <code>__rrshift__</code> , <code>__rand__</code> , <code>__rxor__</code> , <code>__ror__</code>
Operadores bit a bit de atribuição combinada	<code>__ilshift__</code> , <code>__irshift__</code> , <code>__iand__</code> , <code>__ixor__</code> , <code>__ior__</code>



Os operadores reversos são alternativas usadas quando os operandos são trocados (b * a no lugar de a * b), enquanto as atribuições combinadas são formas abreviadas que combinam um operador infixo com a atribuição a uma variável (a = a * b torna-se a *= b). O capítulo 13 explica tanto os operadores reversos quanto as atribuições combinadas em detalhes.

Por que len não é um método?

Fiz essa pergunta ao core developer Raymond Hettinger em 2013, e a chave de sua resposta foi uma citação de The Zen of Python (<https://www.python.org/doc/humor/#the-zen-of-python>): “practicality beats purity” (a praticidade supera a pureza). Na seção “Como os métodos especiais são usados” na página 33, descrevo como len(x) é executado rapidamente quando x é uma instância de um tipo embutido. Nenhum método é chamado para os objetos embutidos do CPython: o tamanho é simplesmente lido de um campo de uma struct em C. Obter a quantidade de itens de uma coleção é uma operação comum e precisa funcionar de modo eficiente para tipos básicos e tão diversificados como str, list, memoryview e assim por diante.

Em outras palavras, `len` não é chamado como um método porque ele recebe um tratamento especial como parte do modelo de dados do Python, assim como `abs`. Porém, graças ao método especial `__len__`, você pode fazer `len` funcionar também com seus objetos personalizados. Esse é um compromisso justo entre a necessidade de ter objetos embutidos eficientes e a consistência da linguagem. Também de *The Zen of Python*: “Special cases aren’t special enough to break the rules” (Casos especiais não são especiais o suficiente para infringir as regras).



Se pensarmos em `abs` e em `len` como operadores unários, poderemos nos sentir mais inclinados a desculpar sua aparência funcional, em oposição à sintaxe da chamada de método esperada em uma linguagem orientada a objetos. De fato, a linguagem ABC – ancestral direta de Python que foi pioneira em muitas de suas funcionalidades – tinha um operador `#` que era equivalente a `len` (escrito como `#s`). Quando usado como operador infixo, escrito como `x#s`, as ocorrências de `x` em `s` eram contadas, o que, em Python, fazemos com `s.count(x)` para qualquer sequência `s`.

Resumo do capítulo

Implementando métodos especiais, seus objetos podem se comportar como os tipos embutidos, possibilitando o estilo expressivo de programação considerado pythônico pela comunidade.

Um requisito básico para um objeto Python é produzir representações úteis de si mesmo em forma de string, uma para debugging ou logging e outra para a apresentação aos usuários finais. É por isso que os métodos especiais `__repr__` e `__str__` fazem parte do modelo de dados.

Emular sequências, conforme mostrado no exemplo com `FrenchDeck`, é uma das aplicações mais comuns dos métodos especiais. Tirar o maior proveito dos tipos para sequências é o assunto do capítulo 2, e a implementação de suas próprias sequências será discutida no capítulo 10, em que criaremos uma extensão multidimensional da classe `Vector`.

Graças à sobrecarga de operadores, Python oferece um conjunto rico de tipos numéricos que vão dos tipos embutidos até `decimal.Decimal` e `fractions.Fraction`; todos aceitam operadores aritméticos infixos. A implementação de operadores, incluindo os operadores reversos e as atribuições combinadas, será mostrada no capítulo 13 por meio de melhorias ao exemplo com `Vector`.

O uso e a implementação da maioria dos demais métodos especiais do modelo de dados de Python serão discutidos ao longo deste livro.

Leituras complementares

O capítulo “Data Model” (Modelo de dados, <http://docs.python.org/3/reference/datamodel.html>) de *The Python Language Reference* (Guia de referência à linguagem Python) é a fonte canônica para o assunto deste capítulo e de boa parte deste livro.

O livro *Python in a Nutshell*, 2ª edição (O’Reilly, <http://bit.ly/Python-IAN>) de Alex Martelli contém uma discussão excelente sobre o modelo de dados. Quando escrevi este livro, a edição mais recente do livro *Nutshell* era de 2006 e focava o Python 2.5, porém houve poucas mudanças no modelo de dados desde então, e a descrição de Martelli do funcionamento do acesso aos atributos é a mais bem fundamentada que já vi, sem considerar o código-fonte C propriamente dito do CPython. Martelli também é um colaborador assíduo do Stack Overflow, com mais de cinco mil respostas postadas. Consulte o perfil de seu usuário no Stack Overflow (<http://stackoverflow.com/users/95810/alex-martelli>).

David Beazley escreveu dois livros que abordam o modelo de dados em detalhes no contexto do Python 3 – *Python Essential Reference*, 4ª edição (Addison-Wesley Professional) e *Python Cookbook*, 3ª edição (O’Reilly)³ –, este último, escrito em conjunto com Brian K. Jones.

O livro *The Art of the Metaobject Protocol* (AMOP, MIT Press), de Gregor Kiczales, Jim des Rivieres e Daniel G. Bobrow, explica o conceito de MOP (Metaobject Protocol, ou Protocolo de Metaobjetos), do qual o modelo de dados do Python é um exemplo.

Ponto de vista

Modelo de dados ou modelo de objetos?

O que a documentação do Python chama de Python data model (“Modelo de dados do Python”) a maioria dos autores chama de Python object model (“Modelo de objetos do Python”). Os livros *Python in a Nutshell*, 2ª edição, de Alex Martelli, e *Python Essential Reference*, 4ª edição, de David Beazley, são os melhores livros que discutem o “modelo de dados do Python”, porém eles sempre se referem a ele como o “modelo de objetos”. Na Wikipedia, a primeira definição de modelo de objetos (http://en.wikipedia.org/wiki/Object_model) é: “The properties of objects in general in a specific computer programming language.” (As propriedades dos objetos em geral em uma linguagem específica de programação de computadores.) É a isso que o “modelo de dados do Python” se refere. Neste livro, usarei “modelo de dados” porque a documentação favorece esse termo ao referir-se ao modelo de objetos

3 N.T.: Tradução brasileira publicada pela Novatec (<http://novatec.com.br/livros/python-cookbook/>).

do Python e porque é o título do capítulo de *The Python Language Reference* (Guia de referência à linguagem Python, <https://docs.python.org/3/reference/datamodel.html>), mais relevante às nossas discussões.

Métodos mágicos

A comunidade Ruby chama seu equivalente aos métodos especiais de *métodos mágicos* (magic methods). Muitas pessoas da comunidade Python adotam esse termo também. Acredito que os métodos especiais, na realidade, são o oposto de mágicos. O Python e o Ruby são iguais quanto a este aspecto: ambos oferecem aos usuários um protocolo rico de metaobjetos que não é mágico, mas permite aos usuários tirar proveito das mesmas ferramentas disponíveis aos core developers. Para comparar, considere o JavaScript. Os objetos nessa linguagem têm recursos que são mágicos no sentido que não é possível emulá-los nos objetos definidos pelo próprio usuário. Por exemplo, antes do JavaScript 1.8.5, não era possível definir atributos somente para leitura (read-only) em seus objetos JavaScript, porém alguns objetos embutidos sempre tinham atributos somente para leitura. Em JavaScript, os atributos somente para leitura eram “mágicos”, exigindo poderes sobrenaturais que um usuário da linguagem não tinha até surgir o ECMAScript 5.1 em 2009. O protocolo de metaobjetos do JavaScript está evoluindo, porém, historicamente, tem sido mais limitado do que aquele usado pelo Python e pelo Ruby.

Metaobjetos

The Art of the Metaobject Protocol (AMOP) é o meu título favorito entre os livros de computação. Falando de modo menos subjetivo, o termo *protocolo de metaobjetos* é útil para pensarmos no modelo de dados do Python e em recursos semelhantes de outras linguagens. A palavra *metaobjeto* diz respeito aos objetos que constituem os blocos de construção da própria linguagem. Nesse contexto, *protocolo* é sinônimo de *interface*. Portanto *protocolo de metaobjetos* é um sinônimo elegante para modelo de objetos: uma API para as construções essenciais da linguagem.

Um protocolo de metaobjetos rico permite estender uma linguagem para que ela aceite novos paradigmas de programação. Gregor Kiczales, o primeiro autor do livro AMOP, posteriormente tornou-se pioneiro da programação orientada a aspectos e o primeiro autor do AspectJ, uma extensão de Java que implementa esse paradigma. A programação orientada a aspectos é muito mais simples de implementar em uma linguagem dinâmica como o Python, e diversos frameworks fazem isso, porém o mais importante é o *zope.interface*, que será brevemente discutido na seção “Leituras complementares” na página 389 do capítulo 11.

PARTE II

Estruturas de dados

CAPÍTULO 2

Uma coleção de sequências

Como você deve ter notado, diversas operações mencionadas funcionam igualmente para textos, listas e tabelas. Os textos, as listas e as tabelas são chamados de trens (*trains*). O comando FOR também funciona de modo genérico em trens.¹

— Geurts, Meertens e Pemberton
ABC Programmer's Handbook

Antes de criar a linguagem Python, Guido foi colaborador no projeto ABC – uma pesquisa de dez anos para projetar um ambiente de programação para iniciantes. A linguagem ABC introduziu muitas ideias que atualmente consideramos “pythônicas”: operações padronizadas em sequências, tipos embutidos para tuplas e mapeamentos, estrutura de blocos por indentação, tipagem forte sem declaração de variáveis etc. Não é acidente o fato de Python ser tão amigável ao usuário.

Python herdou o tratamento uniforme de sequências da linguagem ABC. Strings, listas, sequências de bytes, arrays, elementos XML e resultados de consultas a banco de dados compartilham um rico conjunto de operações comuns, incluindo iteração, slicing (fatiamento), ordenação e concatenação.

Entender a variedade de sequências disponíveis em Python evita que reinventemos a roda, e sua interface comum nos inspira a criar APIs que aceitem e tirem proveito dos tipos de sequência existentes e que surgirão no futuro.

A maior parte da discussão neste capítulo se aplica a sequências em geral, do familiar tipo `list` aos tipos `str` e `bytes` que são novidade no Python 3. Tópicos específicos sobre listas, tuplas, arrays e filas também serão discutidos aqui, porém o foco em strings Unicode e sequências de bytes será adiado até o capítulo 4. Além disso, a ideia neste capítulo é discutir os tipos de sequência prontos para uso. Criar seus próprios tipos de sequência será o assunto do capítulo 10.

¹ Leo Geurts, Lambert Meertens e Steven Pemberton, *ABC Programmer's Handbook*, p. 8.

Visão geral das sequências embutidas

A biblioteca-padrão disponibiliza um conjunto rico de tipos de sequência implementados em C:

Sequências container

`list`, `tuple` e `collections.deque` podem armazenar itens de tipos diferentes.

Sequências simples

`str`, `bytes`, `bytearray`, `memoryview` e `array.array` armazenam itens de um só tipo.

As *sequências container* armazenam referências aos objetos que elas contêm, que podem ser de qualquer tipo, enquanto as *sequências simples* armazenam fisicamente o valor de cada item em seu próprio espaço de memória, e não como objetos distintos. Desse modo, as sequências simples são mais compactas, porém estão limitadas ao armazenamento de valores primitivos como caracteres, bytes e números.

Outra maneira de agrupar os tipos de sequência é de acordo com a mutabilidade:

Sequências mutáveis

`list`, `bytearray`, `array.array`, `collections.deque` e `memoryview`

Sequências imutáveis

`tuple`, `str` e `bytes`

A figura 2.1 ajuda a visualizar como as sequências mutáveis diferem das sequências imutáveis, ao mesmo tempo que herdam diversos métodos delas. Observe que os tipos concretos de sequências embutidas não são realmente subclasses das ABCs (abstract base classes, ou classes-base abstratas) `Sequence` e `MutableSequence` representadas, mas assim mesmo as ABCs são úteis como uma formalização das funcionalidades esperadas de um tipo de sequência completo.

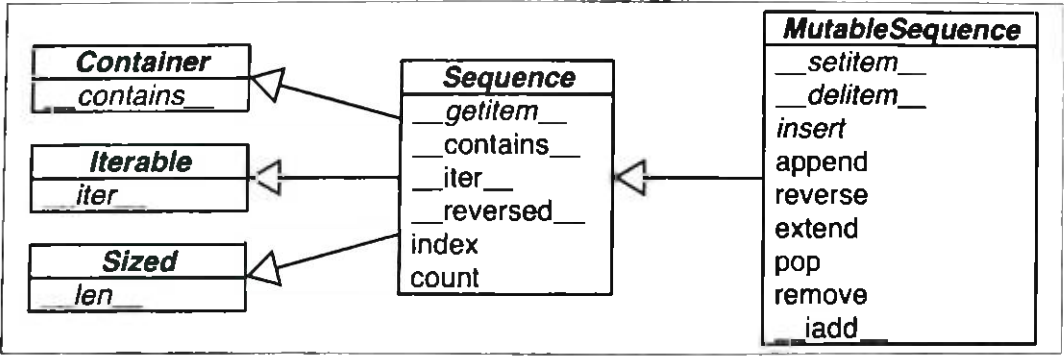


Figura 2.1 – Diagrama de classes UML para algumas classes de `collections.abc` (as superclasses estão à esquerda; as setas de herança apontam das subclasses para as superclasses; os nomes em *itálico* são classes e métodos abstratos).

Ter em mente esses traços comuns – mutável *versus* imutável; container *versus* simples – é útil para extrapolar o que sabemos sobre um tipo de sequência em comparação a outros.

O tipo mais básico de sequência é `list` – um container mutável. Estou certo de que você se sente à vontade para lidar com listas, portanto passaremos diretamente para as *list comprehensions*, que são uma maneira eficiente de criar listas, porém, em algumas equipes, subutilizadas devido à sua sintaxe pouco usual. Dominar as *list comprehensions* abre a porta para as expressões geradoras (*generator expressions*) que – entre outros usos – podem produzir elementos para preencher sequências de qualquer tipo. Ambos os assuntos serão discutidos na próxima seção.

List comprehensions e expressões geradoras

Uma maneira rápida de criar uma sequência é usar uma *list comprehension* (se o alvo for uma `list`) ou uma expressão geradora (para todos os demais tipos de sequência). Se você não estiver usando essas formas sintáticas em seu cotidiano, aposto que estará perdendo oportunidades de escrever um código que é, ao mesmo tempo, mais legível e, geralmente, mais rápido.

Se você duvida de meu argumento de que essas construções sejam “mais legíveis”, continue lendo. Tentarei convencê-lo disso.



Por questões de concisão, muitos programadores Python referem-se às *list comprehensions* como *listcomps* e às expressões geradoras como *genexprs*. Usarei essas palavras também.

List comprehensions e legibilidade

Eis um teste: qual exemplo você acha mais fácil de ler, o exemplo 2.1 ou o exemplo 2.2?

Exemplo 2.1 – Cria uma lista de códigos Unicode (codepoints) a partir de uma string

```
>>> symbols = 'Șț€¥€¤'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Exemplo 2.1 – Cria uma lista de códigos Unicode (codepoints) a partir de uma string, tomada dois

```
>>> symbols = '$ç£¥€¤'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Qualquer pessoa que conheça um pouco de Python poderá ler o exemplo 2.1. No entanto, após ter conhecido as listcomps, acho o exemplo 2.2 mais legível, pois seu propósito é explícito.

Um laço for pode ser usado para realizar várias tarefas diferentes: varrer uma sequência para contar ou selecionar itens, computar agregações (somas, médias) ou executar quaisquer outras tarefas de processamento. O código do exemplo 2.1 cria uma lista. Em comparação, a sintaxe de listcomp foi concebida com um único propósito: criar uma nova lista.

É claro que é possível abusar das list comprehensions para escrever códigos verdadeiramente incompreensíveis. Já vi códigos Python com listcomps usadas somente para repetir um bloco de código por causa de seus efeitos colaterais. Se você não vai fazer nada com a lista gerada, não utilize essa sintaxe. Além disso, procure deixá-la concisa. Se a list comprehension ocupar mais de duas linhas, provavelmente será melhor quebrá-la em partes ou reescrevê-la como um bom e velho laço for. Use o seu bom senso: em Python, assim como em português, não há regras escritas a ferro e fogo para escrever com clareza.



Dica de sintaxe

Em código Python, as quebras de linha são ignoradas entre pares [], {} ou (). Sendo assim, podemos criar listas, listcomps, genexps, dicionários e outras estruturas com múltiplas linhas sem usar \, que é o escape nada elegante para continuação de linha.

As listcomps não deixam mais as variáveis vazarem

Em Python 2.x, as variáveis atribuídas nas cláusulas for nas list comprehensions eram criadas no escopo ao redor, às vezes provocando sérias consequências. Veja a sessão de console com o Python 2.7 a seguir:

```
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 'my precious'
>>> dummy = [x for x in 'ABC']
>>> x
'C'
```

Como podemos ver, o valor inicial de `x` foi afetado. Isso não acontece mais no Python 3.

As list comprehensions, as expressões geradoras e suas parentes próximas, as set e dict comprehensions, agora têm seu próprio escopo local, assim como as funções. As variáveis que recebem valor dentro da expressão são locais, porém as variáveis no escopo ao redor ainda podem ser referenciadas. Melhor ainda, as variáveis locais não mascaram as variáveis do escopo ao redor.

Este código utiliza Python 3:

```
>>> x = 'ABC'
>>> dummy = [ord(x) for x in x]
>>> x ❶
'ABC'
>>> dummy ❷
[65, 66, 67]
>>>
```

- ❶ O valor de `x` foi preservado.
 - ❷ A list comprehension gera a lista esperada.
-

As list comprehensions criam listas a partir de sequências ou de qualquer outro tipo iterável, filtrando e transformando os itens. As funções embutidas `filter` e `map` podem ser empregadas em composição para fazer o mesmo, porém a legibilidade será prejudicada, como veremos a seguir.

Comparação entre listcomps e map/filter

As listcomps fazem tudo que as funções `map` e `filter` fazem, sem os contorcionismos exigidos pelo limitado `lambda` que temos em Python. Considere o exemplo 2.3.

Exemplo 2.3 – A mesma lista criada por uma listcomp e uma composição de map/filter

```
>>> symbols = '$ç€¥€'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

Eu acreditava que `map` e `filter` eram mais rápidas que as `listcomps` equivalentes, porém Alex Martelli mostrou que isso não acontece – pelo menos, não nos exemplos anteriores. O script `02-array-seq/listcomp_speed.py` (<http://bit.ly/1Vm6R3n>) no repositório de código do livro *Python fluente* (<https://github.com/fluentpython/example-code>) é um teste simples de velocidade que compara a `listcomp` com `filter/map`.

Terei mais a dizer sobre `map` e `filter` no capítulo 5. Agora vamos nos voltar ao uso de `listcomps` para calcular produtos cartesianos: uma lista contendo tuplas criadas a partir de todos os itens de duas ou mais listas.

Produtos cartesianos

As `listcomps` podem gerar listas a partir do produto cartesiano de dois ou mais iteráveis. Os itens que compõem o produto cartesiano são tuplas compostas de itens de todos os iteráveis de entrada. A lista resultante tem um tamanho igual aos tamanhos dos iteráveis de entrada multiplicados.

Veja a figura 2.2.

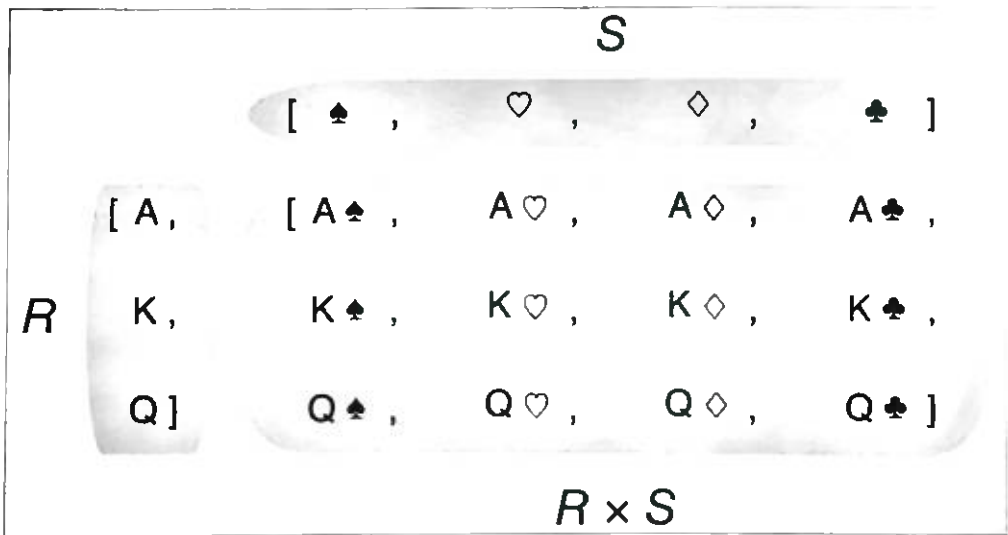


Figura 2.2 – O produto cartesiano de uma sequência de três valores de cartas e uma sequência de quatro naipes resulta em uma sequência de doze pares.

Por exemplo, suponha que você deva gerar uma lista de camisetas disponíveis em duas cores e três tamanhos. O exemplo 2.4 mostra como gerar essa lista usando uma `listcomp`. O resultado contém seis itens.

Exemplo 2.4 – Produto cartesiano usando uma list comprehension

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes] ❶
>>> tshirts
```



```
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors: ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes ❸
...             for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

- ❶ Gera uma lista de tuplas organizadas por cor e, em seguida, por tamanho.
- ❷ Observe como a lista resultante está organizada como se os loops for estivessem aninhados na mesma ordem em que aparecem na listcomp.
- ❸ Para obter os itens organizados por tamanho e, em seguida, pela cor, basta reorganizar as cláusulas for; adicionar uma quebra de linha à listcomp faz com que seja mais fácil ver como o resultado estará ordenado.

No exemplo 1.1 (Capítulo 1), a expressão a seguir foi utilizada para inicializar um conjunto de cartas de baralho com uma lista composta de 52 cartas com todos os 13 valores de cada um dos 4 naipes, agrupados por naipe:

```
self._cards = [Card(rank, suit) for suit in self.suits
               for rank in self.ranks]
```

As listcomps fazem apenas uma coisa: elas criam listas. Para preencher outros tipos de sequência, uma genexp é a opção adequada. A próxima seção contém uma breve descrição do uso de genexps para criar sequências que não sejam listas.

Expressões geradoras

Para inicializar tuplas, arrays e outros tipos de sequência, poderíamos começar também com uma listcomp, porém uma genexp economiza memória, pois ela gera itens um por um usando o protocolo de iteradores, em vez de criar uma lista completa somente para alimentar outro construtor.

As genexps utilizam a mesma sintaxe das listcomps, porém são delimitadas por parênteses, e não por colchetes.

O exemplo 2.5 mostra um uso básico das genexps para criar uma tupla e um array.

Exemplo 2.5 – Inicializando uma tupla e um array a partir de uma expressão geradora

```
>>> symbols = '$ç£¥€π'
>>> tuple(ord(symbol) for symbol in symbols) ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols)) ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

- ❶ Se a expressão geradora for o único argumento em uma chamada de função, não será necessário duplicar os parênteses.
- ❷ O construtor array aceita dois argumentos, portanto os parênteses em torno da expressão geradora são obrigatórios. O primeiro argumento do construtor array define o tipo de armazenamento usado para os números no array, conforme veremos na seção “Arrays” na página 75.

O exemplo 2.6 utiliza uma genexp com um produto cartesiano para exibir uma lista de camisetas de duas cores em três tamanhos. Em comparação com o exemplo 2.4, neste caso, a lista de camisetas com seis itens não é criada na memória: a expressão geradora alimenta o laço for gerando um item de cada vez. Se as duas listas usadas no produto cartesiano tivessem mil itens cada, usar uma expressão geradora evitaria o custo de criar uma lista com um milhão de itens somente para alimentar o laço for.

Exemplo 2.6 – Produto cartesiano em uma expressão geradora

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in ('%s %s' % (c, s) for c in colors for s in sizes): ❶
...     print(tshirt)
...
black S
black M
black L
white S
white M
white L
```

- ❶ A expressão geradora gera itens um por um; uma lista com todas as seis variações de camiseta jamais será construída nesse exemplo.

O capítulo 14 é dedicado a explicar como os geradores (generators) funcionam em detalhes. Nesse exemplo, a ideia foi somente mostrar o uso das expressões geradoras para inicializar sequências que não sejam listas ou gerar uma saída que não precise ser armazenada em memória.

Agora vamos prosseguir para outro tipo fundamental de sequência em Python: a tupla.

Tuplas não são apenas listas imutáveis

Alguns textos introdutórios sobre Python apresentam as tuplas como “listas imutáveis”, porém elas são bem mais do que isso. As tuplas têm dupla função: podem ser usadas como listas imutáveis e também como registros sem nomes de campos. Esse uso às vezes é menosprezado, portanto vamos começar por ele.

Tuplas como registros

Tuplas armazenam registros: cada item da tupla armazena os dados de um campo, e a posição do item na tupla confere o seu significado.

Se pensarmos em uma tupla somente como uma lista imutável, a quantidade e a ordem dos itens poderão ou não ser importantes, dependendo do contexto. Porém, quando usamos uma tupla como uma coleção de campos, a quantidade de itens geralmente será fixa e sua ordem sempre será muito importante.

O exemplo 2.7 mostra tuplas sendo usadas como registros. Observe que, em todas as expressões, reordenar a tupla destruiria as informações, pois o significado de cada item é dado pela sua posição na tupla.

Exemplo 2.7 – Tuplas usadas como registros

```
>>> lax_coordinates = (33.9425, -118.408056) ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32450, 0.66, 8014) ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ❸
...                 ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids): ❹
...     print('%s/%s' % passport) ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids: ❻
...     print(country)
```

...

USA

BRA

ESP

- ❶ Latitude e longitude do Aeroporto Internacional de Los Angeles.
- ❷ Dados sobre Tóquio: nome, ano, população (milhões), mudança na população (%), área (km²).
- ❸ Uma lista de tuplas no formato (country_code, passport_number).
- ❹ À medida que fazemos uma iteração pela lista, o nome `passport` é associado a cada tupla.
- ❺ O operador de formatação `%` entende as tuplas e trata cada item como um campo separado.
- ❻ O laço `for` sabe como obter os itens de uma tupla separadamente – isso é chamado de “desempacotamento” (unpacking). Nesse caso, não estamos interessados no segundo item, portanto ele é atribuído a `_`, que é uma variável comumente usada para capturar valores que não queremos usar.

As tuplas funcionam bem como registros por causa do mecanismo de desempacotamento de tuplas, que será o nosso próximo assunto.

Desempacotamento de tuplas

No exemplo 2.7, atribuímos (`'Tokyo'`, `2003`, `32450`, `0.66`, `8014`) a `city`, `year`, `pop`, `chg`, `area` em uma única instrução. Em seguida, na última linha, o operador `%` atribuiu cada item da tupla `passport` a uma posição da string de formatação do argumento de `print`. Esses são dois exemplos de *desempacotamento de tuplas*.



O desempacotamento de tuplas funciona com qualquer objeto iterável. O único requisito é que o iterável gere exatamente um item por variável na tupla receptora, a menos que você use um asterisco (`*`) para capturar os itens excedentes, conforme explicado na seção “Usando `*` para capturar itens excedentes” na página 55. O termo *desempacotamento de tuplas* é amplamente usado pelos pythonistas, porém a expressão *desempacotamento de iteráveis* está ganhando força, como mostra o título da PEP 3132 – *Extended Iterable Unpacking* (Desempacotamento estendido de iteráveis, <http://python.org/dev/peps/pep-3132/>).

A forma mais visível de desempacotamento de tuplas é a *atribuição paralela* (parallel assignment) – ou seja, atribuir itens de um iterável a uma tupla de variáveis, como podemos ver no exemplo a seguir:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates # desempacotamento de tupla
>>> latitude
33.9425
>>> longitude
-118.408056
```

Uma aplicação elegante do desempacotamento de tuplas consiste em trocar (swap) os valores de duas variáveis sem usar uma variável temporária:

```
>>> b, a = a, b
```

Outro exemplo de desempacotamento de tuplas consiste em prefixar um argumento com um asterisco ao chamar uma função:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

O código anterior também mostra outro uso do desempacotamento de tuplas: permitir que as funções devolvam diversos valores de uma forma conveniente para quem chamou. Por exemplo, a função `os.path.split()` cria uma tupla (`path`, `last_part`) a partir de um caminho do sistema de arquivos:

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/idrsa.pub')
>>> filename
'idrsa.pub'
```

Às vezes, quando estivermos interessados somente em determinadas partes de uma tupla ao fazer o desempacotamento, uma variável descartável como `_` poderá ser usada para ocupar um lugar na tupla, como no exemplo anterior.



Se você escreve softwares internacionalizados, o símbolo `_` não é uma boa variável descartável, pois ele é tradicionalmente usado como um apelido para a função `gettext.gettext`, conforme recomendado na documentação do módulo `gettext` (<http://docs.python.org/3/library/gettext.html>). Em outros casos, `_` é um ótimo nome para uma variável descartável.

Outra maneira de pegar apenas alguns itens ao fazer o desempacotamento em uma tupla é usar `*`, como veremos agora.

Usando * para capturar itens excedentes

Definir parâmetros de função com `*args` para capturar quaisquer argumentos em excesso é um recurso clássico em Python.

Em Python 3, essa ideia foi estendida para ser aplicada também às atribuições paralelas:

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

No contexto da atribuição paralela, o prefixo `*` pode ser aplicado exatamente a uma variável, mas poderá aparecer em qualquer posição:

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

Por fim, um recurso poderoso do desempacotamento de tuplas é o suporte a estruturas aninhadas.

Desempacotamento de tuplas aninhadas

A tupla que receberá uma expressão a ser desempacotada pode ter tuplas aninhadas, por exemplo, `(a, b, (c, d))`, e Python fará “a coisa certa” se a expressão combinar com a estrutura aninhada. O exemplo 2.8 mostra o desempacotamento de tuplas aninhadas em ação.

Exemplo 2.8 – Desempacotando tuplas aninhadas para acessar a longitude

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)), # ❶
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
```

```
]
```

```
print('{:15} | {:^9} | {:^9}'.format('', 'lat.', 'long.'))
fmt = '{:15} | {:9.4f} | {:9.4f}'
for name, cc, pop, (latitude, longitude) in metro_areas: # ❷
    if longitude <= 0: # ❸
        print(fmt.format(name, latitude, longitude))
```

- ❶ Cada tupla armazena um registro com quatro campos, em que o último é um par de coordenadas.
- ❷ Ao atribuir o último campo a uma tupla, desempacotamos as coordenadas.
- ❸ `if longitude <= 0`: limita a saída às áreas metropolitanas do hemisfério ocidental.

A saída do exemplo 2.8 é:

	lat.	long.
Mexico City	19.4333	-99.1333
New York-Newark	40.8086	-74.0204
Sao Paulo	-23.5478	-46.6358



Antes do Python 3, era possível definir funções com tuplas aninhadas nos parâmetros formais (por exemplo, `def fn(a, (b, c), d):`). Isso não é mais aceito nas definições de função no Python 3 por razões práticas explicadas na *PEP 3113 – Removal of Tuple Parameter Unpacking* (Remoção do desempacotamento de tuplas como parâmetros, <http://python.org/dev/peps/pep-3113/>). Para ficar bem claro: nada mudou do ponto de vista dos usuários que chamam uma função. A restrição aplica-se somente à definição de funções.

Do modo como foram concebidas, as tuplas são muito práticas. Porém há um recurso que às vezes faz falta quando as utilizamos como registros: associar nomes aos campos. É por isso que a função `namedtuple` foi criada. Continue lendo.

Tuplas nomeadas

A função `collections.namedtuple` é uma fábrica (factory) que gera subclasses de `tuple` melhoradas com nomes de campos e um nome de classe – o que ajuda no debugging.



As instâncias de uma classe criada com `namedtuple` ocupam exatamente a mesma quantidade de memória que tuplas porque os nomes dos campos são armazenados na classe. Elas usam menos memória que um objeto normal, pois não armazenam atributos em um `__dict__` por instância.

Vamos recordar como criamos a classe `Card` no exemplo 1.1 do capítulo 1:

```
Card = collections.namedtuple('Card', ['rank', 'suit'])
```


O exemplo 2.9 mostra como definir uma tupla nomeada para armazenar informações sobre uma cidade.

Exemplo 2.9 – Definindo e usando uma tupla nomeada

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population coordinates') ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667)) ❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722,
139.691667))
>>> tokyo.population ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

- ❶ Dois parâmetros são necessários para criar uma tupla nomeada: um nome de classe e uma lista de nomes de campos especificados como um iterável de strings ou como uma única string delimitada por espaços.
- ❷ Os dados devem ser passados como argumentos posicionais ao construtor (em comparação, o construtor `tuple` aceita um único iterável).
- ❸ Você pode acessar os campos por nome ou pela posição.

Uma tupla nomeada tem alguns atributos além daqueles herdados de `tuple`. O exemplo 2.10 mostra os mais úteis: o atributo de classe `_fields`, o método de classe `_make(iterable)` e o método de instância `_asdict()`.

Exemplo 2.10 – Atributos e métodos de uma tupla nomeada (continuação do exemplo anterior)

```
>>> City._fields ❶
('name', 'country', 'population', 'coordinates')
>>> LatLong = namedtuple('LatLong', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, LatLong(28.613889, 77.208889))
>>> delhi = City._make(delhi_data) ❷
>>> delhi._asdict() ❸
OrderedDict([('name', 'Delhi NCR'), ('country', 'IN'), ('population',
21.935), ('coordinates', LatLong(lat=28.613889, long=77.208889))])
>>> for key, value in delhi._asdict().items():
    print(key + ': ', value)

name: Delhi NCR
country: IN
```



```
population: 21.935
coordinates: LatLong(lat=28.613889, long=77.208889)
>>>
```

- ❶ `_fields` é uma tupla com os nomes dos campos da classe.
- ❷ `_make()` permite instanciar uma tupla nomeada a partir de um iterável; `City(*delhi_data)` faria o mesmo.
- ❸ `_asdict()` retorna um `collections.OrderedDict` criado a partir da instância da tupla nomeada. Pode ser usado para exibir os dados da cidade de uma forma bem legível.

Agora que exploramos a utilidade das tuplas como registros, podemos considerar a sua função secundária como uma variante imutável do tipo `list`.

Tuplas como listas imutáveis

Ao usar uma `tuple` como uma variação imutável de `list`, é bom saber até que ponto elas são realmente semelhantes. Como podemos ver na tabela 2.1, `tuple` aceita todos os métodos de `list` que não envolvam acrescentar ou remover itens, com uma exceção: tuplas não têm o método `__reversed__`. No entanto esse método existe apenas por otimização; a chamada `reversed(my_tuple)` funciona sem ele.

Tabela 2.1 – Métodos e atributos encontrados em `list` ou em `tuple` (métodos implementados por `object` foram omitidos por questões de concisão)

	list	tuple	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> – concatenação
<code>s.__iadd__(s2)</code>	•		<code>s += s2</code> – concatenação in-place
<code>s.append(e)</code>	•		Concatena um elemento após o último
<code>s.clear()</code>	•		Apaga todos os itens
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		Shallow copy (cópia rasa) da lista
<code>s.count(e)</code>	•	•	Conta as ocorrências de um elemento
<code>s.__delitem__(p)</code>	•		Remove o item da posição <code>p</code>
<code>s.extend(it)</code>	•		Adiciona itens do iterável <code>it</code>
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> – obtém o item de uma posição
<code>s.__getnewargs__()</code>		•	Suporte para serialização otimizada com <code>pickle</code>
<code>s.index(e)</code>	•	•	Encontra a posição da primeira ocorrência de <code>e</code>
<code>s.insert(p, e)</code>	•		Insere o elemento <code>e</code> antes do item na posição <code>p</code>

	list	tuple	
s.__iter__()	●	●	Obtém um iterador
s.__len__()	●	●	len(s) – número de itens
s.__mul__(n)	●	●	s * n – concatenação repetida
s.__imul__(n)	●		s *= n – concatenação repetida in-place
s.__rmul__(n)	●	●	n * s – concatenação repetida (operador reverso) (*)
s.pop([p])	●		Remove e retorna o último item ou, opcionalmente, o item na posição p
s.remove(e)	●		Remove a primeira ocorrência do elemento com o valor de e
s.reverse()	●		Inverte a ordem dos itens in-place
s.__reversed__()	●		Obtém um iterador para percorrer os itens do último para o primeiro
s.__setitem__(p, e)	●		s[p] = e – coloca e na posição p sobrescrevendo o item existente
s.sort([key], [reverse])	●		Ordena itens in-place com os argumentos nomeados opcionais key e reverse

(*) Os operadores reversos serão explicados no capítulo 13.

Todo programador Python sabe que as sequências podem ser fatiadas usando a sintaxe `s[a:b]`. Vamos agora discutir alguns fatos menos conhecidos sobre o fatiamento (slicing).

Fatiamento

Um recurso comum de `list`, `tuple`, `str` e de todos os tipos de sequência em Python é o suporte às operações de fatiamento (slicing), que são mais poderosas do que a maioria das pessoas imagina.

Nesta seção, descreveremos o *uso* dessas formas avançadas de fatiamento. Sua implementação em uma classe definida pelo usuário será discutida no capítulo 10, seguindo nossa filosofia de discutir as classes prontas para usar nesta parte do livro e a criação de novas classes na Parte IV.

Por que as fatias e os intervalos excluem o último item

A convenção pythônica de excluir o último item de fatias e de intervalos (`range`) funciona bem com a indexação baseada em zero usada em Python, C e em diversas outras linguagens. Algumas características convenientes da convenção são:

- É fácil ver o tamanho de uma fatia ou de um intervalo quando somente a posição final é especificada: tanto `range(3)` quanto `my_list[:3]` geram três itens.

- É fácil calcular o tamanho de uma fatia ou de um intervalo quando o início e o fim são especificados: basta subtrair `stop - start`.
- É fácil separar uma sequência em duas partes em qualquer índice `x`, sem que haja sobreposição: basta fazer `my_list[:x]` e `my_list[x:]`. Por exemplo:

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2] # quebrar no 2
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3] # quebrar no 3
[10, 20, 30]
>>> l[3:]
[40, 50, 60]
```

Porém os melhores argumentos para essa convenção foram escritos por Edsger W. Dijkstra, cientista de computação holandês (veja a última referência na seção “Leituras complementares” na página 88).

Agora vamos dar uma olhada mais detalhada em como o Python interpreta a notação de fatiamento.

Objetos slice

Isto não é nenhum segredo, mas vale a pena repetir somente por garantia: você pode usar `s[a:b:c]` para especificar um salto ou um passo `c`, fazendo a fatia resultante pular itens. O passo também pode ser negativo, o que fará os itens serem devolvidos na ordem inversa. Três exemplos tornarão isso mais claro:

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

Outro exemplo foi mostrado no capítulo 1, quando usamos `deck[12::13]` para obter todos os ases do baralho antes de este ser embaralhado:

```
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
 Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

A notação `a:b:c` é válida somente entre `[]` quando usada como o operador de indexação e resultará em um objeto do tipo `slice`: `slice(a, b, c)`. Como veremos na seção “Como funciona o fatiamento” na página 324, para avaliar a expressão `seq[start:stop:step]`, Python chama `seq.__getitem__(slice(start, stop, step))`. Mesmo que você não esteja implementando seus próprios tipos de sequência, conhecer os objetos `slice` será conveniente porque ele permite atribuir nomes às fatias, assim como as planilhas permitem nomear intervalos de células.

Suponha que você precise analisar os dados de um arquivo de texto de colunas fixas, como a fatura mostrada no exemplo 2.11. Em vez de encher o seu código com fatias fixas, você pode nomeá-las. Veja como isso torna o laço `for` no final do exemplo mais legível.

Exemplo 2.11 – Itens de cada linha de um arquivo simples de fatura

```
>>> invoice = """
... 0.....6.....40.....52...55.....
... 1909  Pimoroni PiBrella                $17.50   3    $52.50
... 1489  6mm Tactile Switch x20           $4.95   2    $9.90
... 1510  Panavise Jr. - PV-201            $28.00   1   $28.00
... 1601  PiTFT Mini Kit 320x240          $34.95   1   $34.95
... """
>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
>>> QUANTITY = slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
...     print(item[UNIT_PRICE], item[DESCRIPTION])
...
$17.50  Pimoroni PiBrella
$4.95   6mm Tactile Switch x20
$28.00  Panavise Jr. - PV-201
$34.95  PiTFT Mini Kit 320x240
```

Voltaremos aos objetos `slice` quando discutirmos a criação de suas próprias coleções na seção “Vector, tomada #2: uma sequência que permite fatiamento” na página 323. Enquanto isso, do ponto de vista de um usuário, o fatiamento inclui funcionalidades adicionais como fatias multidimensionais e notação de reticência (...). Continue lendo.

Fatiamento multidimensional e reticências

O operador `[]` também pode aceitar vários índices ou fatias separados por vírgulas. Isso é usado, por exemplo, no pacote externo NumPy, em que os itens de um `numpy.ndarray` bidimensional podem ser acessados com a sintaxe `a[i, j]` e uma fatia bidimensional pode ser obtida com uma expressão como `a[m:n, k:l]`. O exemplo 2.22 mais adiante neste capítulo mostra o uso dessa notação. Os métodos especiais `__getitem__` e `__setitem__` que tratam o operador `[]` simplesmente recebem os índices em `a[i, j]` como uma tupla. Em outras palavras, para avaliar `a[i, j]`, o Python chama `a.__getitem__((i, j))`.

Os tipos embutidos de sequência em Python são unidimensionais, portanto aceitam somente um índice ou uma fatia, e não uma tupla com eles.

As reticências – escritas com três pontos finais (...) e não com ... (Unicode U+2026) – são reconhecidas como um elemento sintático (token) pelo parser do Python. É um alias para o objeto `Ellipsis`, que é a única instância da classe `ellipsis`.² Desse modo, ele pode ser passado como um argumento para funções e como parte da especificação de uma fatia, assim `f(a, ..., z)` ou assim `a[i:...]`. O NumPy usa ... como uma forma abreviada no fatiamento de arrays de várias dimensões; por exemplo, se `x` for um array de quatro dimensões, `x[i, ...]` é uma forma abreviada para `x[i, :, :, :]`. Veja o *Tentative NumPy Tutorial* (Tutorial Provisório para o NumPy, http://wiki.scipy.org/Tentative_NumPy_Tutorial) para aprender mais sobre esse assunto.

Ao escrever estas linhas, eu desconheço qualquer uso de `Ellipsis` ou de índices e fatias multidimensionais na biblioteca-padrão de Python. Se você encontrar algum, me conte. Esses recursos sintáticos existem para dar suporte a tipos definidos pelo usuário e a extensões como o NumPy.

As fatias não são úteis apenas para extrair informações de sequências; elas também podem ser usadas para alterar sequências mutáveis in-place – ou seja, sem precisar recriá-las.

Atribuição de valores a fatias

Sequências mutáveis podem ser enxertadas, extirpadas e modificadas in-place usando a notação de fatias do lado esquerdo de uma atribuição ou como alvo de um comando `del`. Os próximos exemplos darão uma ideia do poder dessa notação:

```
>>> l = list(range(10))
>>> l
```

2 Não, eu não inverti esses nomes: o nome da classe `ellipsis` realmente é todo escrito com letras minúsculas, e a instância é um objeto embutido chamado `Ellipsis`, assim como o tipo `bool` só tem letras minúsculas, porém suas instâncias são `True` e `False`.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3:2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100 ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

- ❶ Quando o alvo da atribuição é uma fatia, o lado direito deve ser um objeto iterável, mesmo que tenha apenas um item.

Todos sabem que a concatenação é uma operação comum com sequências de qualquer tipo. Qualquer texto introdutório de Python explica o uso de `+` e de `*` para essa finalidade, porém há alguns detalhes sutis no modo como eles funcionam, que serão discutidos a seguir.

Usando `+` e `*` com sequências

Programadores Python esperam que as sequências aceitem `+` e `*`. Geralmente ambos operandos de `+` devem ser do mesmo tipo de sequência, e nenhum deles será modificado, mas uma nova sequência do mesmo tipo será criada como resultado da concatenação.

Para concatenar diversas cópias da mesma sequência, multiplique-a por um inteiro. Aqui também uma nova sequência será criada:

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

Tanto `+` quanto `*` sempre criam um novo objeto e jamais alteram seus operandos.



Tome cuidado com expressões como `a * n` quando `a` for uma sequência contendo itens mutáveis, pois o resultado poderá surpreender você. Por exemplo, tentar inicializar uma lista de listas como `my_list = [[]] * 3` resultará em uma lista com três referências à mesma lista interna, o que, provavelmente, não deve ser o que você quer fazer.

Na próxima seção, discutiremos as armadilhas existentes na tentativa de usar `*` para inicializar uma lista de listas.

Criando listas de listas

Às vezes, precisamos inicializar uma lista com determinado número de listas aninhadas – por exemplo, para distribuir alunos em uma lista de equipes ou para representar quadrados em um jogo de tabuleiro. A melhor maneira de fazer isso é usar uma *list comprehension*, como mostra o exemplo 2.12.

Exemplo 2.12 – Uma lista com três listas de tamanho 3 pode representar um tabuleiro de jogo da velha

```
>>> board = [['_'] * 3 for i in range(3)] ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X' ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

❶ Cria uma lista de três listas com três itens cada. Inspeciona a estrutura.

❷ Coloca uma marca na linha 1, coluna 2, e verifica o resultado.

Um atalho tentador, porém incorreto, é fazer o que mostra o exemplo 2.13.

Exemplo 2.13 – Uma lista com três referências à mesma lista é inútil

```
>>> weird_board = [['_'] * 3] * 3 ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = 'O' ❷
>>> weird_board
[['_', '_', 'O'], ['_', '_', 'O'], ['_', '_', 'O']]
```

❶ A lista externa é composta de três referências à mesma lista interna. Enquanto ela permanece inalterada, tudo parece estar correto.

❷ Colocar uma marca na linha 1, coluna 2 mostra que todas as linhas são aliases que se referem ao mesmo objeto.

O problema com o exemplo 2.13 está no fato de que, essencialmente, ele se comporta como o código a seguir:

```
row = ['_'] * 3
board = []
for i in range(3):
    board.append(row) ❶
```

❶ O mesmo row é concatenado três vezes a board.

Por outro lado, a list comprehension do exemplo 2.12 é equivalente ao código a seguir:

```
>>> board = []
>>> for i in range(3):
...     row = ['_'] * 3 # ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board # ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

❶ Cada iteração cria um novo row e o concatena a board.

❷ Somente a linha 2 é alterada, conforme esperado.



Se o problema ou a solução desta seção não estiverem claros para você, não se preocupe. O capítulo 8 foi escrito para esclarecer o funcionamento e as armadilhas associados às referências e aos objetos mutáveis.

Até agora, discutimos o uso dos operadores simples + e * com sequências, porém há também os operadores += e *=, que geram resultados bem diferentes, dependendo da mutabilidade da sequência-alvo. A seção a seguir explica como isso funciona.

Atribuições combinadas e sequências

Os operadores de atribuição combinada += e *= comportam-se de modo bastante diferente conforme o primeiro operando. Para simplificar a discussão, focaremos primeiro na adição combinada (+=), porém os conceitos também se aplicam a *= e a outros operadores de atribuição combinada.

O método especial que faz `+=` funcionar é `__iadd__` (de “in-place addition”, ou soma in-place). No entanto, se `__iadd__` não estiver implementado, o Python usará `__add__` como alternativa. Considere esta expressão simples:

```
>>> a += b
```

Se `a` implementar `__iadd__`, ele será chamado. No caso de sequências mutáveis (por exemplo, `list`, `bytearray`, `array.array`), `a` será alterado in-place (ou seja, o efeito será semelhante a `a.extend(b)`). No entanto, quando `a` não implementar `__iadd__`, a expressão `a += b` terá o mesmo efeito que `a = a + b`: a expressão `a + b` será avaliada antes, gerando um novo objeto que, então, será associado a `a`. Em outras palavras, a identidade do objeto associado a `a` poderá ou não mudar, de acordo com a disponibilidade de `__iadd__`.

Em geral, para sequências mutáveis, supor que `__iadd__` está implementado e que `+=` ocorre in-place é uma boa aposta. Para sequências imutáveis, é claro que não há como isso acontecer.

O que acabei de escrever sobre `+=` também se aplica a `*=`, que é implementado por meio de `__imul__`. Os métodos especiais `__iadd__` e `__imul__` serão discutidos no capítulo 13.

Aqui está uma demonstração de `*=` com uma sequência mutável e, em seguida, com uma sequência imutável:

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800 ❶
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800 ❷
>>> t = (1, 2, 3)
>>> id(t)
4312681568 ❸
>>> t *= 2
>>> id(t)
4301348296 ❹
```

❶ ID da lista inicial.

❷ Após a multiplicação, a lista será o mesmo objeto, com novos itens acrescentados.

❸ ID da tupla inicial.

❹ Após a multiplicação, uma nova tupla foi criada.

A concatenação repetida de sequências imutáveis é ineficiente, pois, em vez de simplesmente concatenar novos itens, o interpretador precisa copiar toda a sequência-alvo para criar uma nova sequência com os novos itens concatenados.³

Vimos casos de uso comuns para `+=`. A próxima seção mostra um caso bizarro que enfatiza o verdadeiro significado da palavra “imutável” no contexto das tuplas.

O enigma da atribuição `+=`

Procure responder sem usar o console: qual é o resultado da avaliação das duas expressões do exemplo 2.14?⁴

Exemplo 2.14 – Uma charada

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

O que acontecerá a seguir? Escolha a resposta mais adequada:

- a. `t` torna-se `(1, 2, [30, 40, 50, 60])`.
- b. `TypeError` é gerado com a mensagem `'tuple' object does not support item assignment` (objeto `'tuple'` não suporta atribuição de item).
- c. Nenhuma das opções.
- d. Tanto **a** quanto **b** estão corretas.

Quando vi isso, eu tive certeza de que a resposta era **b**, mas, na verdade, é **d**, “Tanto **a** quanto **b** estão corretas”! O exemplo 2.15 mostra a saída real de um console do Python 3.4 (na verdade, o resultado é o mesmo em um console do Python 2.7).⁵

Exemplo 2.15 – O resultado inesperado: o item `t2` é alterado e uma exceção é gerada

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

3 `str` é uma exceção a essa descrição. Pelo fato de a criação de strings com `+=` em laços ser bem comum por aí, o CPython está otimizado para esse caso de uso. As instâncias de `str` são alocadas em memória com espaço extra, de modo que a concatenação não exigirá uma cópia da string completa todas as vezes.

4 Agradeço a Leonardo Rochael e a Cesar Kawakami por terem compartilhado essa charada na Conferência Brasileira de Python de 2013.

5 Um leitor sugeriu que a operação do exemplo pode ser realizada com `t[2].extend([50,60])`, sem erros. Sabemos disso, porém o propósito do exemplo é discutir o comportamento inusitado do operador `+=`.

O Online Python Tutor (<http://www.pythontutor.com/>) é uma ferramenta online incrível para visualizar como Python funciona em detalhes. A figura 2.3 mostra uma composição com duas imagens de tela contendo os estados inicial e final da tupla `t` do exemplo 2.15.

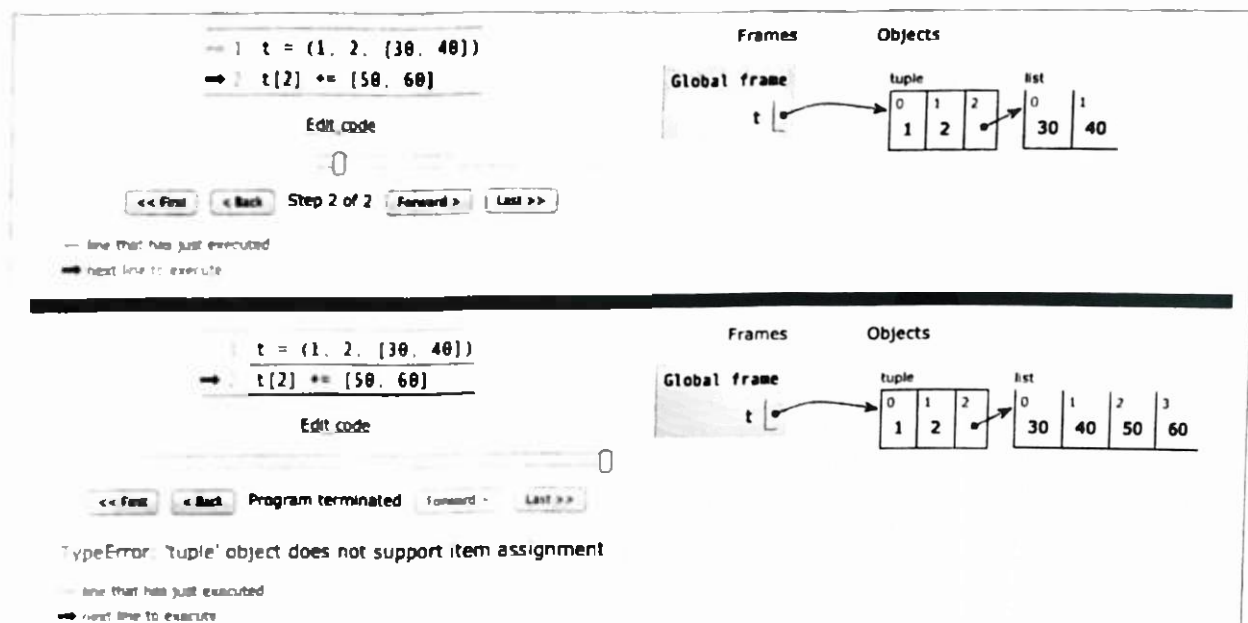


Figura 2.3 – Estados inicial e final do enigma da atribuição de tupla (diagrama gerado pelo Online Python Tutor).

Se você observar o bytecode gerado pelo Python para a expressão `s[a] += b` (Exemplo 2.16), ficará mais claro ver como isso acontece.

Exemplo 2.16 – Bytecode para a expressão `s[a] += b`

```
>>> dis.dis('s[a] += b')
1          0 LOAD_NAME                0 (s)
          3 LOAD_NAME                1 (a)
          6 DUP_TOP_TWO
          7 BINARY_SUBSCR              ❶
          8 LOAD_NAME                2 (b)
         11 INPLACE_ADD                ❷
         12 ROT_THREE
         13 STORE_SUBSCR              ❸
         14 LOAD_CONST                0 (None)
         17 RETURN_VALUE
```

- ❶ Coloca o valor de `s[a]` no TOS (Top Of Stack, ou Topo da Pilha).
- ❷ Executa `TOS += b`. Isso funciona quando TOS refere-se a um objeto mutável (é uma lista no exemplo 2.15).
- ❸ Faz a atribuição `s[a] = TOS`. Isso falha se `s` é imutável (a tupla `t` no exemplo 2.15).

Esse exemplo é um caso realmente peculiar – em quinze anos usando Python, jamais vi esse comportamento estranho prejudicar alguém na vida real.

Tiro três lições disso:

- Colocar itens mutáveis em tuplas não é uma boa ideia.
- Uma atribuição combinada não é uma operação atômica – acabamos de vê-la lançando uma exceção após ter feito parte de seu trabalho.
- Inspeccionar bytecodes Python não é muito difícil e, com frequência, ajudará a ver o que está acontecendo internamente.

Após ter observado as sutilezas no uso de `+` e `*` para concatenação, podemos mudar de assunto e discutir outra operação essencial com sequências: a ordenação.

list.sort e a função embutida sorted

O método `list.sort` ordena uma lista in-place – isto é, sem criar uma cópia. Ele devolve `None` para nos lembrar de que o objeto-alvo é alterado e que não foi criada uma nova lista. Essa é uma convenção importante da API de Python: as funções ou os métodos que alteram um objeto in-place devolvem `None` para deixar claro a quem chamou que o objeto em si foi alterado e que nenhum objeto novo foi criado. O mesmo comportamento pode ser visto, por exemplo, na função `random.shuffle`.



A convenção de devolver `None` para indicar alterações in place tem uma desvantagem: não é possível chamar esses métodos em cascata. Por outro lado, os métodos que devolvem novos objetos (por exemplo, todos os métodos de `str`) podem ser chamados em cascata no estilo conhecido como “interface fluente”. Consulte o artigo “Fluent interface” (http://en.wikipedia.org/wiki/Fluent_interface) na Wikipedia em inglês para saber mais sobre esse assunto.

Por outro lado, a função embutida `sorted` cria uma nova lista e a retorna. Na verdade ela aceita qualquer objeto iterável como argumento, incluindo sequências imutáveis e geradores (veja o capítulo 14). Independentemente do tipo de iterável fornecido a `sorted`, uma nova lista recém-criada sempre será devolvida.

Tanto `list.sort` quanto `sorted` aceitam dois parâmetros opcionais que, quando usados, precisam ser nomeados:

`reverse`

Se `for True`, os itens serão devolvidos em ordem decrescente (ou seja, invertendo a comparação dos itens). O default é `False`.

key

Uma função de um só argumento aplicada a cada item para gerar a sua chave de ordenação. Por exemplo, ao ordenar uma lista de strings, `key=str.lower` pode ser usado para ordenar sem levar em consideração a diferença entre maiúsculas e minúsculas (case-insensitive), e `key=len` ordena as strings de acordo com o número de caracteres. O default é a função identidade (ou seja, os próprios itens são comparados).



O parâmetro nomeado `key` opcional também pode ser usado nas funções embutidas `min()` e `max()`, além de outras funções da biblioteca-padrão (por exemplo, `itertools.groupby()` e `heapq.nlargest()`).

Eis alguns exemplos para deixar mais claro o uso dessas funções e dos argumentos nomeados⁶:

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry'] ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple'] ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry'] ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple'] ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❻
>>> fruits.sort() ❼
>>> fruits
['apple', 'banana', 'grape', 'raspberry'] ❽
```

- ❶ Gera uma nova lista de strings em ordem alfabética.
- ❷ Ao inspecionar a lista original, vemos que ela não foi alterada.
- ❸ Essa é simplesmente uma inversão da ordem alfabética.
- ❹ Uma nova lista de strings, agora ordenada de acordo com o tamanho. Como o algoritmo de ordenação é estável, “grape” e “apple”, ambas de tamanho 5, permanecem na ordem original.

⁶ Os exemplos também mostram que o Timsort – o algoritmo de ordenação usado em Python – é estável (ou seja, preserva a ordem relativa dos itens comparados como iguais). O Timsort será discutido posteriormente na caixa de texto “Ponto de vista” no final deste capítulo.

- ⑤ Essas são as strings ordenadas em ordem decrescente de tamanho. Não é o inverso do resultado anterior porque a ordenação é estável, portanto novamente “grape” aparece antes de “apple”.
- ⑥ Até agora, a ordem da lista `fruits` original não mudou.
- ⑦ Ordena a lista in-place e devolve o valor `None` (não exibido pelo console).
- ⑧ Agora `fruits` está ordenada.

Depois que suas sequências estiverem ordenadas, elas podem ser pesquisadas com eficiência. Felizmente, o algoritmo-padrão de busca binária já está disponível no módulo `bisect` da biblioteca-padrão de Python. Discutiremos seus recursos essenciais a seguir, incluindo a conveniente função `bisect.insort`, que pode ser usada para garantir que suas sequências ordenadas permaneçam assim.

Administrando sequências ordenadas com `bisect`

O módulo `bisect` oferece duas funções principais – `bisect` e `insort` – que usam o algoritmo de busca binária para encontrar rapidamente e inserir itens em qualquer sequência ordenada.

Pesquisando com `bisect`

`bisect(haystack, needle)` faz uma busca binária de `needle` em `haystack` – que deverá ser uma sequência ordenada – para localizar a posição em que `needle` poderá ser inserido, ao mesmo tempo que `haystack` permanece em ordem crescente⁷. Em outras palavras, todos os itens que aparecerem até essa posição são menores ou iguais a `needle`. Você poderia usar o resultado de `bisect(haystack, needle)` como o argumento `index` de `haystack.insert(index, needle)` – no entanto usar `insort` realiza os dois passos e é mais rápido.



Raymond Hettinger – um colaborador Python muito produtivo – tem uma receita de `SortedCollection` (<http://bit.ly/1Vm6WEa>) que tira proveito do módulo `bisect`, porém é mais simples de usar do que essas funções independentes.

O exemplo 2.17 utiliza um conjunto cuidadosamente selecionado de “agulhas” para mostrar as posições de inserção retornadas por `bisect`. A saída é mostrada na figura 2.4.

⁷ N.T. Nesta seção, as variáveis `haystack` (palheiro) e `needle` (agulha) foram escolhidas em referência à expressão “procurar uma agulha em um palheiro”. O `haystack` é o espaço onde se faz a busca, e `needle` é o objeto a ser encontrado.

Exemplo 2.17 – bisect encontra os pontos de inserção para os itens em uma sequência ordenada

```
import bisect
import sys

HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 20, 21, 23, 23, 26, 29, 30]
NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]

ROW_FMT = '{0:2d} @ {1:2d} {2}{0:<2d}'

def demo(bisect_fn):
    for needle in reversed(NEEDLES):
        position = bisect_fn(HAYSTACK, needle) ❶
        offset = position * ' | ' ❷
        print(ROW_FMT.format(needle, position, offset)) ❸

if __name__ == '__main__':
    if sys.argv[-1] == 'left': ❹
        bisect_fn = bisect.bisect_left
    else:
        bisect_fn = bisect.bisect

    print('DEMO:', bisect_fn.__name__) ❺
    print('haystack ->', ' '.join('%2d' % n for n in HAYSTACK))
    demo(bisect_fn)
```

- ❶ Usa a função bisect escolhida para obter o ponto de inserção.
- ❷ Cria um padrão de barras verticais proporcionais a offset.
- ❸ Exibe as linhas formatadas mostrando o valor de needle e o ponto de inserção.
- ❹ Escolhe a função bisect a ser usada de acordo com o último argumento da linha de comando.
- ❺ Exibe o cabeçalho com o nome da função selecionada.

```
02-array-seq/ $ python3 bisect_demo.py
DEMO: bisect
haystack -> 1 4 5 6 8 12 15 20 21 23 23 26 29 30
31 @ 14 | | | | | | | | | | | | | | 31
30 @ 14 | | | | | | | | | | | | | | 30
29 @ 13 | | | | | | | | | | | | | | 29
23 @ 11 | | | | | | | | | | | | 23
22 @ 9 | | | | | | | | | | 22
10 @ 5 | | | | | 10
8 @ 5 | | | | 8
5 @ 3 | | 5
2 @ 1 | 2
1 @ 1 | 1
0 @ 0 |
```

Figura 2.4 – Saída do exemplo 2.17 usando bisect – cada linha começa com a notação needle @ posição, e o valor de needle aparece novamente abaixo de seu ponto de inserção em haystack.

O comportamento de `bisect` pode ser ajustado de duas maneiras.

Primeiro, um par de argumentos opcionais, `lo` e `hi`, permite restringir a região da sequência a ser pesquisada na inserção. O default de `lo` é 0 e de `hi` é o `len()` da sequência.

Segundo, `bisect`, na verdade, é um alias de `bisect_right`, e há uma função-irmã chamada `bisect_left`. A diferença é perceptível somente quando a agulha for igual a um item da lista: `bisect_right` retorna um ponto de inserção após o item existente, enquanto `bisect_left` retorna a posição do item existente, de modo que a inserção ocorrerá antes dele. Com tipos simples como `int`, isso não faz nenhuma diferença; porém, se a sequência contiver objetos que sejam distintos apesar de serem comparados como iguais, isso poderá ser relevante. Por exemplo, `1` e `1.0` são valores distintos, porém `1 == 1.0` é `True`. A figura 2.5 mostra o resultado do uso de `bisect_left`.

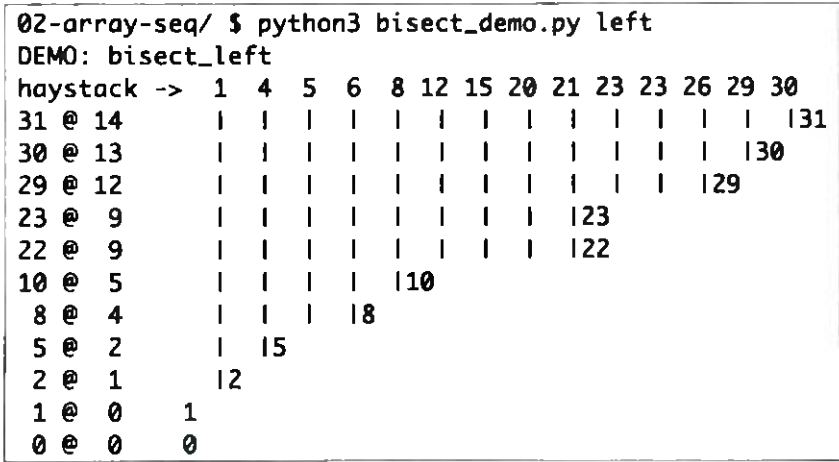


Figura 2.5 – Saída do exemplo 2.17 com o uso de `bisect_left` (compare com a figura 2.4 e observe os pontos de inserção para os valores 1, 8, 23, 29 e 30 à esquerda dos mesmos números em `haystack`).

Uma aplicação interessante de `bisect` está na pesquisa de valores numéricos em tabelas – por exemplo, converter pontuações em provas para notas em forma de letras, como mostra o exemplo 2.18.

Exemplo 2.18 – Dada a pontuação em uma prova, `grade` retorna a nota correspondente na forma de uma letra

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect.bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

O código no exemplo 2.18 foi extraído da documentação do módulo `bisect` (<https://docs.python.org/3/library/bisect.html>), que também lista as funções para usar `bisect` como um substituto mais rápido para o método `index` ao fazer pesquisas em sequências longas e ordenadas de números.

Essas funções são usadas não só para pesquisas, mas também para inserir itens em sequências ordenadas, como mostra a seção a seguir.

Inserção com `bisect.insort`

Ordenar é uma tarefa custosa, portanto, após ter uma sequência ordenada, é bom mantê-la dessa maneira. É para isso que `bisect.insort` foi criada.

`insort(seq, item)` insere `item` em `seq` de modo a manter `seq` em ordem crescente. Veja o exemplo 2.19 e a sua saída na figura 2.6.

Exemplo 2.19 – `insort` mantém uma sequência ordenada sempre ordenada

```
import bisect
import random

SIZE = 7

random.seed(1729)

my_list = []
for i in range(SIZE):
    new_item = random.randrange(SIZE*2)
    bisect.insort(my_list, new_item)
    print('%2d ->' % new_item, my_list)
```

```
02-array-seq/ $ python3 bisect_insor.py
10 -> [10]
0 -> [0, 10]
6 -> [0, 6, 10]
8 -> [0, 6, 8, 10]
7 -> [0, 6, 7, 8, 10]
2 -> [0, 2, 6, 7, 8, 10]
10 -> [0, 2, 6, 7, 8, 10, 10]
```

Figura 2.6 – Saída do exemplo 2.19.

Assim como `bisect`, `insort` aceita argumentos `lo`, `hi` opcionais para limitar a pesquisa a uma subsequência. Há também uma variação `insort_left` que utiliza `bisect_left` para encontrar os pontos de inserção.

Muito do que vimos até agora neste capítulo aplica-se a sequências em geral, e não apenas a listas ou tuplas. Os programadores Python, às vezes, utilizam o tipo `list` exageradamente porque ele é prático – sei que já fiz isso. Se você estiver lidando com listas de números, os arrays são a melhor opção. O resto do capítulo será dedicado a sequências que podem substituir listas com vantagens.

Quando uma lista não é a resposta

O tipo `list` é flexível e fácil de usar, porém, dependendo dos requisitos específicos, há opções melhores. Por exemplo, se houver necessidade de armazenar dez milhões de valores de ponto flutuante, um `array` será muito mais eficiente, pois um `array` não armazena objetos `float` completos, mas apenas os bytes compactos que representam seus valores de máquina – assim como um `array` na linguagem C. Por outro lado, se você adicionar e remover itens constantemente das extremidades de uma lista como uma estrutura de dados FIFO ou LIFO, um `deque` (double-ended queue, ou fila dupla) será mais rápido.



Se o seu código executa muitas verificações de continência (por exemplo, `item in my_collection`), considere o uso de um `set` para `my_collection`, especialmente se um grande número de itens for armazenado. Os conjuntos (sets) são otimizados para uma verificação rápida de pertinência. Porém eles não são sequências (seu conteúdo não está ordenado), e serão discutidos no capítulo 3.

No restante deste capítulo, discutiremos os tipos mutáveis de sequência que podem substituir as listas em muitos casos, começando pelos `arrays`.

Arrays

Se a lista contiver somente números, um `array.array` será mais eficiente que uma `list`: ele aceita todas as operações de sequências mutáveis (incluindo `.pop`, `.insert` e `.extend`), além de ter métodos adicionais para carregar e salvar rapidamente, por exemplo, `.frombytes` e `.tofile`.

Um `array` Python é tão enxuto quanto um `array` C. Ao criar um `array`, você deve fornecer um `typecode` (código de tipo), isto é, uma letra para determinar o tipo C subjacente usado para armazenar cada item do `array`. Por exemplo, `b` é o `typecode` para `signed char`. Se um `array('b')` for criado, cada item será armazenado em um único byte e será interpretado como um inteiro de -128 a 127 . Para sequências longas de números, isso economizará bastante memória. Além disso, o Python não deixará você colocar nenhum número que não corresponda ao tipo do `array`.

O exemplo 2.20 mostra como criar, salvar e carregar um `array` com dez milhões de números de ponto flutuante aleatórios.

Exemplo 2.20 – Criando, salvando e carregando um `array` grande de números de ponto flutuante

```
>>> from array import array ❶
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7))) ❷
```

```

>>> floats[-1] ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp) ❹
>>> fp.close()
>>> floats2 = array('d') ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7) ❻
>>> fp.close()
>>> floats2[-1] ❼
0.07802343889111107
>>> floats2 == floats ❽
True

```

- ❶ Importa o tipo array.
- ❷ Cria um array de números de ponto flutuante com dupla precisão (typecode 'd') a partir de qualquer objeto iterável – nesse caso, é uma expressão geradora.
- ❸ Inspecciona o último número do array.
- ❹ Salva o array em um arquivo binário.
- ❺ Cria um array vazio de doubles.
- ❻ Lê dez milhões de números do arquivo binário.
- ❼ Inspecciona o último número do array.
- ❽ Verifica se o conteúdo dos arrays coincide.

Como podemos ver, `array.tofile` e `array.fromfile` são fáceis de usar. Se testar o exemplo, você perceberá que eles também são muito rápidos. Um breve experimento mostra que demora aproximadamente 0,1 segundo para `array.fromfile` carregar dez milhões de números de ponto flutuante com dupla precisão de um arquivo binário criado com `array.tofile`. Isso é quase 60 vezes mais rápido que ler os números de um arquivo-texto, que também envolve fazer parse de cada linha com a função embutida `float`. Salvar com `array.tofile` é aproximadamente sete vezes mais rápido que escrever um número de ponto flutuante por linha em um arquivo-texto. Além do mais, o tamanho do arquivo binário com dez milhões de doubles é igual a 80.000.000 bytes (8 bytes por double, sem desperdício), enquanto o arquivo-texto com os mesmos números tem 181.515.739 bytes.

Para o caso específico de arrays numéricos que representem dados binários, por exemplo, imagens raster, Python tem os tipos `bytes` e `bytearray`, que serão discutidos no capítulo 4.



Outra maneira rápida e mais flexível de salvar dados numéricos é usar o módulo `pickle` (<http://bit.ly/py-pickle>) para serialização de objetos. Salvar um array de números de ponto flutuante com `pickle.dump` é quase tão rápido quanto salvar com `array.tofile` – no entanto `pickle` trata quase todos os tipos embutidos, incluindo números complex, coleções aninhadas e até mesmo instâncias de classes definidas pelo usuário automaticamente (se elas não tiverem uma implementação muito complicada).

Concluimos esta seção sobre arrays com a tabela 2.2 que apresenta uma comparação dos recursos de `list` e de `array.array`.

Tabela 2.2 – Métodos e atributos encontrados em `list` ou em `array` (métodos obsoletos de `array` e aqueles implementados por `object` foram omitidos por questões de concisão)

	list	array	
<code>s.__add__(s2)</code>	●	●	<code>s + s2</code> – concatenação
<code>s.__iadd__(s2)</code>	●	●	<code>s += s2</code> – concatenação in-place
<code>s.append(e)</code>	●	●	Concatena um elemento após o último
<code>s.byteswap()</code>		●	Troca os bytes de todos os itens do array para uma conversão de endianness
<code>s.clear()</code>	●		Apaga todos os itens
<code>s.__contains__(e)</code>	●	●	<code>e in s</code>
<code>s.copy()</code>	●		Cópia rasa (shallow copy) da lista
<code>s.__copy__()</code>		●	Suporte para <code>copy.copy</code>
<code>s.count(e)</code>	●	●	Conta as ocorrências de um elemento
<code>s.__deepcopy__()</code>		●	Suporte otimizado para <code>copy.deepcopy</code>
<code>s.__delitem__(p)</code>	●	●	Remove o item da posição <code>p</code>
<code>s.extend(it)</code>	●	●	Concatena itens do iterável <code>it</code>
<code>s.frombytes(b)</code>		●	Concatena itens da sequência de bytes interpretada como valores de máquina compactos.
<code>s.fromfile(f, n)</code>		●	Concatena <code>n</code> itens do arquivo binário <code>f</code> interpretado como valores de máquina compactos.
<code>s.fromlist(l)</code>		●	Concatena itens da lista; se algum deles provocar um <code>TypeError</code> , nenhum valor será concatenado
<code>s.__getitem__(p)</code>	●	●	<code>s[p]</code> – obtém o item de uma posição
<code>s.index(e)</code>	●	●	Encontra a posição da primeira ocorrência de <code>e</code>
<code>s.insert(p, e)</code>	●	●	Insere o elemento <code>e</code> antes do item na posição <code>p</code>
<code>s.itemsize</code>		●	Tamanho em bytes de cada item do array
<code>s.__iter__()</code>	●	●	Obtém um iterador

	list	array	
s.__len__()	●	●	len(s) – número de itens
s.__mul__(n)	●	●	s * n – concatenação repetida
s.__imul__(n)	●	●	s *= n – concatenação repetida in-place
s.__rmul__(n)	●	●	n * s – concatenação repetida invertida ^a
s.pop([p])	●	●	Remove e retorna um item na posição p (default: último)
s.remove(e)	●	●	Remove a primeira ocorrência do elemento e de acordo com o valor
s.reverse()	●	●	Inverte a ordem dos itens in-place
s.__reversed__()	●		Obtém um iterador para varrer os itens do último para o primeiro
s.__setitem__(p, e)	●	●	s[p] = e – coloca e na posição p sobrescrevendo o item existente
s.sort([key], [reverse])	●		Ordena itens in-place com os argumentos nomeados opcionais key e reverse
s.tobytes()		●	Devolve os itens como valores de máquina compactos em um objeto bytes
s.tofile(f)		●	Salva itens como valores de máquina compactos em um arquivo binário f
s.tolist()		●	Devolve os itens como objetos numéricos em uma list
s.typecode		●	String de um caractere que identifica o tipo dos itens na linguagem C

a Os operadores invertidos serão explicados no capítulo 13.



Até Python 3.4, o tipo array não tem um método sort in place como list.sort(). Se for necessário ordenar um array, use a função sorted para recriá-lo em ordem:

```
a = array.array(a.typecode, sorted(a))
```

Para manter um array ordenado enquanto itens são adicionados a ele, use a função bisect.insort (conforme vimos na seção “Inserção com bisect.insort” na página 74).

Se você faz muitas tarefas com arrays e não conhece a `memoryview`, está perdendo algo. Veja o tópico a seguir.

Memory Views

A classe `memoryview` embutida é um tipo de sequência de memória compartilhada que permite lidar com fatias de arrays sem copiar os bytes. Foi inspirada na biblioteca NumPy (que será discutida brevemente na seção “NumPy e SciPy” na página 80). Travis Oliphant, autor principal da NumPy, responde à pergunta “Quando devemos usar uma `memoryview`?” da seguinte maneira:

Uma `memoryview` é essencialmente uma estrutura de array NumPy genérica no próprio Python (sem a matemática). Ela permite compartilhar memória entre estruturas de dados (informações como imagens PIL, bancos de dados SQLite, arrays NumPy etc.) sem fazer uma cópia inicial. Isso é muito importante para conjuntos grandes de dados.

Usando uma notação semelhante à do módulo `array`, o método `memoryview.cast` permite alterar o modo como vários bytes são lidos ou escritos como unidades sem mover os dados por aí (como o operador `cast` em C). `memoryview.cast` devolve outro objeto `memoryview`, sempre compartilhando a mesma memória.

Observe o exemplo 2.21 para ver como alterar um único byte de um array de inteiros de 16 bits.

Exemplo 2.21 – Alterando o valor de um item do array ao mudar um de seus bytes

```
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers) ❶
>>> len(memv)
5
>>> memv[0] ❷
-2
>>> memv_oct = memv.cast('B') ❸
>>> memv_oct.tolist() ❹
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4 ❺
>>> numbers
array('h', [-2, -1, 1024, 1, 2]) ❻
```

- ❶ Cria uma `memoryview` a partir de um array de 5 short signed integers (typecode 'h').
- ❷ `memv` vê os mesmos 5 itens do array.
- ❸ Cria `memv_oct` ao fazer o casting dos elementos de `memv` para o typecode 'B' (unsigned char).
- ❹ Exporta os elementos de `memv_oct` na forma de uma lista para inspeção.
- ❺ Atribui o valor 4 ao byte de offset 5.
- ❻ Observe a mudança em `numbers`: um 4 no byte mais significativo de um inteiro de dois bytes sem sinal é 1024.

Veremos outro pequeno exemplo com `memoryview` no contexto de manipulações de sequências binárias com `struct` (Capítulo 4; Exemplo 4.4).

Enquanto isso, se estiver fazendo processamentos numéricos sofisticados em arrays, você deverá usar as bibliotecas NumPy e SciPy. Daremos uma olhada rapidamente nelas agora.

NumPy e SciPy

Ao longo deste livro, fiz questão de enfatizar o que já existe na biblioteca-padrão de Python para que você possa tirar o máximo proveito desses recursos. Porém NumPy e SciPy são tão incríveis que um desvio se faz necessário.

Para operações sofisticadas com arrays e matrizes, NumPy e SciPy são o motivo pelo qual Python tornou-se uma linguagem importante em aplicações de computação científica. A NumPy implementa tipos para arrays e matrizes multidimensionais e homogêneos que armazenam não só números como também registros definidos pelo usuário, além de oferecer operações eficientes aplicadas a todos os elementos de uma só vez.

A SciPy é uma biblioteca criada com base em NumPy e oferece muitos algoritmos de computação científica, de álgebra linear a cálculo numérico e estatísticas. A SciPy é rápida e confiável, pois aproveita a base de código C e Fortran amplamente utilizada do *Netlib Repository* (<http://www.netlib.org/>). Em outras palavras, SciPy oferece o melhor de dois mundos aos cientistas: um console interativo e APIs Python de alto nível, juntamente com funções extremamente eficientes para processamento numérico, otimizadas em C e em Fortran.

Como uma demonstração bem rápida, o exemplo 2.22 mostra algumas operações básicas com arrays bidimensionais no NumPy.

Exemplo 2.22 – Operações básicas com linhas e colunas em um `numpy.ndarray`

```
>>> import numpy ❶
>>> a = numpy.arange(12) ❷
>>> a
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape ❸
(12,)
>>> a.shape = 3, 4 ❹
>>> a
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
>>> a[2] ❺
array([ 8, 9, 10, 11])
>>> a[2, 1] ❻
9
>>> a[:, 1] ❼
array([1, 5, 9]) ❽
```



```
>>> a.transpose()
array([[ 0, 4, 8],
       [ 1, 5, 9],
       [ 2, 6, 10],
       [ 3, 7, 11]])
```

- ❶ Importa o Numpy depois que estiver instalado (ele não faz parte da biblioteca-padrão do Python).
- ❷ Cria e inspeciona um `numpy.ndarray` com inteiros de 0 a 11.
- ❸ Inspeciona as dimensões do array: é um array unidimensional com 12 elementos.
- ❹ Muda a forma do array, acrescentando uma dimensão, e inspeciona o resultado em seguida.
- ❺ Obtém a linha no índice 2.
- ❻ Obtém o elemento no índice 2, 1.
- ❼ Obtém a coluna no índice 1.
- ❽ Cria um novo array por transposição (trocando as colunas pelas linhas).

O NumPy também trata operações de alto nível para carregar, salvar e realizar operações em todos os elementos de um `numpy.ndarray`:

```
>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt') ❶
>>> floats[-3:] ❷
array([ 3016362.69195522, 535281.10514262, 4566560.44373946])
>>> floats *= .5 ❸
>>> floats[-3:]
array([ 1508181.34597761, 267640.55257131, 2283280.22186973])
>>> from time import perf_counter as pc ❹
>>> t0 = pc(); floats /= 3; pc() - t0 ❺
0.03690556302899495
>>> numpy.save('floats-10M', floats) ❻
>>> floats2 = numpy.load('floats-10M.npy', 'r+') ❼
>>> floats2 *= 6
>>> floats2[-3:] ❽
memmap([ 3016362.69195522, 535281.10514262, 4566560.44373946])
```

- ❶ Carrega dez milhões de números de ponto flutuante a partir de um arquivo-texto.
- ❷ Utiliza a notação de slicing de sequência para inspecionar os três últimos números.
- ❸ Multiplica todos os elementos do array `floats` por .5 e inspeciona os três últimos elementos novamente.

- ④ Importa o timer de medição de desempenho de alta resolução (disponível desde o Python 3.3).
- ⑤ Divide todos os elementos por 3; o tempo decorrido para dez milhões de números de ponto flutuante é menor que 40 milissegundos.
- ⑥ Salva o array em um arquivo binário *.npy*.
- ⑦ Carrega os dados como um arquivo mapeado em memória para outro array; isso permite um processamento eficiente dos slices do array, mesmo que ele não caiba totalmente na memória.
- ⑧ Inspecciona os três últimos elementos após multiplicar todos os elementos por 6.



Instalar NumPy e SciPy a partir do código-fonte não é fácil. A página *Installing the SciPy Stack* (Instalando a pilha SciPy, <http://www.scipy.org/install.html>) em SciPy.org recomenda usar as distribuições Python científicas especiais como Anaconda, Enthought Canopy e WinPython, entre outras. São downloads grandes, porém estão prontos para uso. Os usuários das distribuições GNU/Linux populares normalmente poderão encontrar NumPy e SciPy nos repositórios de pacotes padrão. Por exemplo, instalá-los no Debian ou no Ubuntu é simples assim:

```
$ sudo apt-get install python-numpy python-sciPy
```

Isso foi apenas um aperitivo. NumPy e SciPy são bibliotecas formidáveis e constituem a base de outras ferramentas incríveis como as bibliotecas de análise de dados Pandas (<http://pandas.pydata.org>) e Blaze (<http://blaze.pydata.org/en/latest/>), que oferecem tipos eficientes de arrays, capazes de armazenar dados não numéricos, assim como funções para importar/exportar em vários formatos diferentes (por exemplo, *.csv*, *.xls*, dumps SQL, HDF5 etc.). Esses pacotes merecem livros inteiros sobre eles. Este não é um desses livros. Entretanto nenhuma descrição geral das sequências em Python estaria completa sem ao menos uma olhada rápida nos arrays NumPy.

Após termos visto as sequências homogêneas – `array.array` e arrays NumPy –, veremos agora um conjunto totalmente diferente de substitutos para a boa e velha `list`: as filas (queues).

Deques e outras filas

Os métodos `.append` e `.pop` tornam uma `list` utilizável como uma pilha ou uma fila (se usar `.append` e `.pop(0)`, você terá um comportamento de LIFO). No entanto inserir e remover itens da esquerda de uma lista (a extremidade com índice 0) é custoso, pois a lista toda será deslocada.

A classe `collections.deque` é uma fila dupla thread-safe (segura para threads), criada para proporcionar inserção e remoção rápidas de ambas as extremidades. Ela também é a opção adequada se houver necessidade de manter uma lista dos “últimos itens vistos” ou algo assim, pois um deque pode ser limitado – ou seja, criado com um tamanho máximo – e, quando estiver cheio, os itens serão descartados da extremidade oposta quando novos itens forem adicionados. O exemplo 2.23 mostra algumas operações típicas realizadas em um deque.

Exemplo 2.23 – Trabalhando com um deque

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3) ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1) ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33]) ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40]) ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

- ❶ O argumento `maxlen` opcional define o número máximo de itens permitido nessa instância de `deque`; isso define um atributo de instância `maxlen` somente para leitura.
- ❷ Fazer a rotação com `n > 0` retira os itens da extremidade direita e os insere na esquerda; quando `n < 0`, os itens serão retirados da esquerda e concatenados à direita.
- ❸ Concatenar em um deque que esteja cheio (`len(d) == d.maxlen`) faz com que os itens da outra extremidade sejam descartados; observe, na próxima linha, que `0` foi descartado.
- ❹ Adicionar três itens à direita remove os valores `-1`, `1` e `2` mais à esquerda.
- ❺ Observe que `extendleft(iter)` funciona pela adição sucessiva de cada item do argumento `iter` à esquerda do deque; desse modo, a posição final dos itens estará invertida.

A tabela 2.3 compara os métodos que são específicos de `list` e de `deque` (eliminando aqueles que também aparecem em `object`).

Tabela 2.3 – Métodos implementados em `list` ou em `deque` (aqueles implementados em `object` foram omitidos por questões de concisão)

	list	deque	
<code>s.__add__(s2)</code>	●		<code>s + s2</code> – concatenação
<code>s.__iadd__(s2)</code>	●	●	<code>s += s2</code> – concatenação in-place
<code>s.append(e)</code>	●	●	Concatena um elemento à direita (após o último)
<code>s.appendleft(e)</code>		●	Concatena um elemento à esquerda (antes do primeiro)
<code>s.clear()</code>	●	●	Apaga todos os itens
<code>s.__contains__(e)</code>	●		<code>e in s</code>
<code>s.copy()</code>	●		Shallow copy (cópia rasa) da lista
<code>s.__copy__()</code>		●	Suporte para <code>copy.copy</code> (shallow copy, ou cópia rasa)
<code>s.count(e)</code>	●	●	Conta as ocorrências de um elemento
<code>s.__delitem__(p)</code>	●	●	Remove o item da posição <code>p</code>
<code>s.extend(i)</code>	●	●	Concatena itens do iterável <code>i</code> à direita
<code>s.extendleft(i)</code>		●	Adiciona itens do iterável <code>i</code> à esquerda
<code>s.__getitem__(p)</code>	●	●	<code>s[p]</code> – obtém o item de uma posição
<code>s.index(e)</code>	●		Encontra a posição da primeira ocorrência de <code>e</code>
<code>s.insert(p, e)</code>	●		Insere o elemento <code>e</code> antes do item na posição <code>p</code>
<code>s.__iter__()</code>	●	●	Obtém um iterador
<code>s.__len__()</code>	●	●	<code>len(s)</code> – número de itens
<code>s.__mul__(n)</code>	●		<code>s * n</code> – concatenação repetida
<code>s.__imul__(n)</code>	●		<code>s *= n</code> – concatenação repetida in-place
<code>s.__rmul__(n)</code>	●		<code>n * s</code> – concatenação repetida invertida (*)
<code>s.pop()</code>	●	●	Remove e devolve o último item (**)
<code>s.popleft()</code>		●	Remove e devolve o primeiro item
<code>s.remove(e)</code>	●	●	Remove a primeira ocorrência do elemento <code>e</code> de acordo com o valor
<code>s.reverse()</code>	●	●	Inverte a ordem dos itens in-place
<code>s.__reversed__()</code>	●	●	Obtém o iterador para varrer os itens do último para o primeiro
<code>s.rotate(n)</code>		●	Mova <code>n</code> itens de uma extremidade para a outra
<code>s.__setitem__(p, e)</code>	●	●	<code>s[p] = e</code> – coloca <code>e</code> na posição <code>p</code> sobrescrevendo o item existente
<code>s.sort([key], [reverse])</code>	●		Ordena itens in place com os argumentos nomeados opcionais <code>key</code> e <code>reverse</code>

Os operadores invertidos serão explicados no capítulo 13.

`a_list.pop(p)` permite remover da posição `p`, porém `deque` não aceita essa opção.

Observe que `deque` implementa a maioria dos métodos de `list`, além de acrescentar alguns métodos específicos ao seu design, como `popleft` e `rotate`. Contudo há um custo oculto: remover itens do meio de um `deque` não é tão rápido assim. Ele é realmente otimizado para inserção e remoção de itens das extremidades.

As operações `append` e `popleft` são atômicas, portanto `deque` é seguro para ser usado como uma fila LIFO em aplicações multithreaded (com várias threads) sem a necessidade de usar locks (travas).

Além de `deque`, outros pacotes da biblioteca-padrão do Python implementam filas:

`queue`

Oferece classes sincronizadas (isto é, thread-safe) `Queue`, `LifoQueue` e `PriorityQueue`, usadas para comunicação segura entre threads. Todas as três classes podem ser limitadas especificando um argumento `maxsize` maior que 0 no construtor. No entanto elas não descartam itens para criar espaço como é feito por `deque`. Em vez disso, quando a fila estiver cheia, a inserção de um novo item causará um bloqueio – ou seja, ela esperará até que outra thread crie espaço removendo um item da fila, que é útil para limitar o número de threads ativas.

`multiprocessing`

Implementa a sua própria `Queue` limitada, muito semelhante a `queue.Queue`, porém com design voltado para a comunicação entre processos. Uma `multiprocessing.JoinableQueue` especializada também está disponível para facilitar o gerenciamento de tarefas.

`asyncio`

Recém-adicionado no Python 3.4, `asyncio` oferece `Queue`, `LifoQueue`, `PriorityQueue` e `JoinableQueue` com APIs inspiradas nas classes contidas nos módulos `queue` e `multiprocessing`, porém adaptadas para administrar tarefas em programação assíncrona.

`heapq`

Em oposição aos três módulos anteriores, `heapq` não implementa uma classe de fila, porém provê funções como `heappush` e `heappop`, que permitem usar uma sequência mutável como uma fila heap ou uma fila priorizada.

Com isso, concluímos nossa descrição geral das alternativas ao tipo `list` e também a nossa exploração dos tipos relacionados a sequências em geral – exceto pelas particularidades de `str` e das sequências binárias, que terão seu próprio capítulo (Capítulo 4).

Resumo do capítulo

Dominar os tipos de sequência da biblioteca-padrão é um pré-requisito para escrever um código Python conciso, eficaz, eficiente e idiomático.

As sequências Python geralmente são classificadas como mutáveis ou imutáveis, porém também é conveniente considerar um eixo diferente: sequências simples e sequências container. As primeiras são mais compactas, rápidas e fáceis de usar, porém estão limitadas ao armazenamento de dados atômicos como números, caracteres e bytes. As sequências container são mais flexíveis, porém podem causar surpresas quando armazenam objetos mutáveis, portanto é preciso tomar cuidado para usá-las corretamente com estruturas de dados aninhadas.

As list comprehensions e as expressões geradoras (generator expressions) são notações poderosas para criar e inicializar sequências. Se você ainda não se sente à vontade com elas, reserve um tempo para dominar o seu uso básico. Não é difícil, logo você será conquistado por elas.

As tuplas em Python têm dois papéis: como registros com campos sem nome e como listas imutáveis. Quando uma tupla é usada como um registro, o desempacotamento de tuplas é a maneira mais segura e legível de acessar os campos. A nova sintaxe `*` torna o desempacotamento de tuplas melhor ainda ao permitir que seja mais fácil ignorar alguns campos e lidar com campos opcionais. A função `namedtuple` não é tão nova, porém merece mais atenção: como as tuplas normais, as tuplas nomeadas apresentam pouco overhead por instância, ao mesmo tempo que proporcionam um acesso conveniente aos campos pelo nome e um prático método `._asdict()` para exportar o registro como um `OrderedDict`.

O fatiamento de sequências é um recurso favorito da sintaxe de Python e é mais poderoso do que muitas pessoas se dão conta. O fatiamento multidimensional e a notação de reticências (...) – conforme usadas no NumPy – também podem ser aceitos em sequências definidas pelo usuário. A atribuição a fatias é uma maneira muito expressiva de alterar sequências mutáveis.

A concatenação repetida como em `seq * n` é conveniente e, se tivermos cuidado, pode ser usada para inicializar listas de listas contendo itens imutáveis. A atribuição combinada com `+=` e `*=` comporta-se de maneira diferente para sequências mutáveis e imutáveis. No último caso, esses operadores necessariamente criam novas sequências. Porém, se a sequência-alvo for mutável, normalmente ela será alterada in-place – mas nem sempre, dependendo do modo como a sequência for implementada.

O método `sort` e a função embutida `sorted` são fáceis de usar e são flexíveis, graças ao argumento opcional `key` aceito por elas, com uma função para computar o critério de ordenação. A propósito, `key` também pode ser usado com as funções embutidas `min` e

`max`. Para manter uma sequência ordenada, sempre insira itens usando `bisect.insort`; para fazer uma busca eficiente em uma sequência ordenada, utilize `bisect.bisect`.

Além das listas e das tuplas, a biblioteca-padrão de Python oferece `array.array`. Embora NumPy e SciPy não façam parte da biblioteca-padrão, se você fizer algum tipo de processamento numérico em conjuntos grandes de dados, você poderá ir muito longe estudando essas bibliotecas, mesmo que estude somente uma parte.

Encerramos analisando `collections.deque`, que é versátil e thread-safe, comparando a sua API com a de `list` na tabela 2.3 e mencionando outras implementações de fila na biblioteca-padrão.

Leituras complementares

O capítulo 1 – “Data Structures” (Estruturas de dados) – de *Python Cookbook*, 3ª edição (O’Reilly)⁸ de David Beazley e Brian K. Jones apresenta diversas receitas que focam sequências, incluindo a “Recipe 1.11. Naming a Slice” (Receita 1.11 – Dar nome a uma fatia), por meio da qual aprendi o truque de atribuir slices a variáveis para melhorar a legibilidade, conforme vimos em nosso exemplo 2.11.

A segunda edição de *Python Cookbook* foi escrita para Python 2.4, porém a maior parte de seu código funciona em Python 3, e muitas das receitas dos capítulos 5 e 6 lidam com sequências. O livro foi editado por Alex Martelli, Anna Martelli Ravenscroft e David Ascher e inclui contribuições de dezenas de pythonistas. A terceira edição foi reescrita do zero e foca mais a semântica da linguagem – particularmente o que mudou no Python 3 –, enquanto o volume anterior dá ênfase às questões pragmáticas (ou seja, o modo de aplicar a linguagem em problemas do mundo real). Apesar de algumas das soluções da segunda edição não serem mais a melhor abordagem, sinceramente, acho que vale a pena ter ambas as edições de *Python Cookbook* à mão.

O *Sorting HOWTO* oficial do Python (<http://docs.python.org/3/howto/sorting.html>) apresenta diversos exemplos de truques sofisticados com `sorted` e `list.sort`.

A PEP 3132 – *Extended Iterable Unpacking* (Desempacotamento estendido de iteráveis, <http://python.org/dev/peps/pep-3132/>) é a fonte canônica para ler sobre o novo uso de `*extra` como alvo em atribuições paralelas. Se quiser ter um vislumbre da evolução do Python, *Missing *-unpacking generalizations* (<http://bugs.python.org/issue2292>) é uma sugestão no bug tracker (sistema de monitoramento de bugs) que propõe um uso mais amplo ainda da notação de desempacotamento de iteráveis. A PEP 448 – *Additional Unpacking Generalizations* (Generalizações adicionais para desempacotamento, <https://www.python.org/dev/peps/pep-0448/>) resultou de discussões sobre esse problema.

8 N.T.: Tradução brasileira publicada pela Novatec Editora (<http://novatec.com.br/livros/python-cookbook/>).

Na época em que esta obra foi escrita, parecia ser bem provável que as alterações propostas fossem incluídas no Python, talvez na versão 3.5.

O post de blog “Less Copies in Python with the Buffer Protocol and memoryviews” (Menos cópias em Python com o protocolo de buffer e as memoryviews, <http://bit.ly/IVm6K7Y>), de Eli Bendersky, inclui um breve tutorial sobre `memoryview`.

Há diversos livros no mercado que discutem o NumPy, até alguns que não mencionam “NumPy” no título. *Python for Data Analysis* (O’Reilly, <http://bit.ly/py-data-analysis>) de Wes McKinney é um desses títulos.

Os cientistas amam a combinação de um prompt interativo com a eficácia do NumPy e do SciPy a ponto de terem desenvolvido o IPython, um substituto extremamente poderoso para o console do Python que também disponibiliza uma GUI, uma ferramenta gráfica integrada para criação de gráficos inline, suporte para *literate programming* (programação literária, ou seja, entrelaçamento de texto e código) e renderização para PDF. As sessões multimídia interativas do IPython podem até mesmo ser compartilhadas por meio de HTTP como notebooks IPython. Veja imagens de tela e vídeos em *The IPython Notebook* (<http://ipython.org/notebook.html>). O IPython é tão interessante que, em 2012, seus core developers – a maioria dos quais é composta de pesquisadores da University of California em Berkeley – receberam 1,15 milhão de dólares da Sloan Foundation para investir em melhorias que foram implementadas no período de 2013 a 2014.

O *The Python Standard Library* (Biblioteca-Padrão do Python), seção 8.3. *collections – Container datatypes* (<https://docs.python.org/3/library/collections.html>), inclui pequenos exemplos e receitas práticas usando `deque` (e outras coleções).

A melhor defesa da convenção em Python de excluir do último item em intervalos e fatias foi escrita pelo próprio Edsger W. Dijkstra em um pequeno comunicado intitulado “Why Numbering Should Start at Zero” (Por que a numeração deve iniciar em zero, <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html>). O tema desse comunicado é notação matemática, porém é relevante para programadores Python porque o professor Dijkstra explica, com rigor e humor, por que a sequência 2, 3, ..., 12 sempre deve ser expressa como $2 \leq i < 13$. Todas as demais convenções razoáveis são refutadas, assim como a ideia de deixar que cada usuário escolha uma convenção. O título refere-se à indexação baseada em zero, porém o comunicado, na verdade, é sobre o motivo pelo qual é desejável que `'ABCDE'[1:3]` signifique `'BC'`, e não `'BCD'`, e por que faz todo sentido escrever 2, 3, ..., 12 como `range(2, 13)`. (A propósito, o comunicado é uma nota escrita à mão, porém a caligrafia é bonita e totalmente legível. Alguém deveria criar uma fonte Dijkstra – eu compraria.)

Pontos de vista

A natureza das tuplas

Em 2012, apresentei um pôster sobre a linguagem ABC na PyCon US. Antes de criar Python, Guido havia trabalhado no interpretador ABC e, sendo assim, veio ver o meu pôster. Entre outros assuntos, falamos dos *compounds* de ABC, que, claramente, são os predecessores das tuplas em Python. Os compounds também aceitam atribuição paralela e são usados como chaves compostas em dicionários (ou em *tabelas* no jargão da ABC). No entanto os compounds não são sequências. Eles não são iteráveis e não podemos obter um campo pelo índice, muito menos fatiar. Um compound é tratado como um todo, ou você pode extrair os campos individuais usando a atribuição paralela – nada mais.

Disse a Guido que essas limitações deixam muito claro o propósito principal dos compounds: eles são apenas registros sem nomes de campo. Sua resposta: “Fazer as tuplas se comportarem como sequências foi um hack.”

Isso mostra a abordagem pragmática que torna o Python muito melhor e mais bem-sucedido que ABC. Do ponto de vista do implementador da linguagem, fazer as tuplas se comportarem como sequências tem um custo baixo. Como resultado, as tuplas podem não ser tão “conceitualmente puras” quanto os compounds, porém temos muitas outras maneiras de usá-las. Elas podem até mesmo ser usadas como listas imutáveis!

É realmente conveniente ter listas imutáveis na linguagem, mesmo que seu tipo não seja chamado de `frozenset`, mas seja realmente uma tupla fazendo de conta que é uma sequência.

“Elegância gera simplicidade”

O uso da sintaxe `*extra` para atribuir vários itens a um parâmetro começou com as definições de função há muito tempo (tenho um livro sobre o Python 1.4 de 1996 que discutia isso). A partir do Python 1.6, a forma `*extra` pode ser usada no contexto de chamadas de função para desempacotar um iterável em vários argumentos – uma operação complementar. Isso é elegante, faz sentido do ponto de vista intuitivo e fez a função `apply` se tornar redundante (ela não existe mais). Atualmente, com o Python 3, a notação `*extra` também funciona à esquerda de atribuições paralelas para capturar itens excedentes, melhorando o que já era um recurso prático da linguagem.

Com cada uma dessas alterações, a linguagem se tornou mais flexível, mais consistente e mais simples ao mesmo tempo. “Elegance begets simplicity” (Elegância gera simplicidade) é o lema de minha camiseta favorita da PyCon de Chicago em 2009. Ela é decorada com uma imagem de Bruce Eckel que representa o hexagrama 22 do I Ching, 賁(bì), “Adorno”, às vezes traduzido como “Graça” ou “Beleza”.

Comparação entre sequências simples e containers

Para enfatizar os diferentes modelos de memória dos tipos de sequência, usei os termos *sequência container* e *sequência simples* (*flat sequence* no original em inglês). A palavra “container” aparece na documentação do *Data Model* (Modelo de dados, <https://docs.python.org/3/reference/datamodel.html#objects-values-and-types>):

Alguns objetos contêm referências a outros objetos; eles são chamados de containers.

Usei o termo “sequência container” para ser específico, pois há containers em Python que não são sequências, como `dict` e `set`. As sequências container podem ser aninhadas porque podem conter objetos de qualquer tipo, incluindo seus próprios tipos.

Por outro lado, as *sequências simples* são tipos de sequência que não podem ser aninhados, pois armazenam somente tipos atômicos simples como inteiros, números de ponto flutuante ou caracteres.

Adotei o termo *sequência simples* porque precisava de um termo que contrastasse com “sequência container”. Não sou capaz de citar uma referência para dar suporte ao uso de *sequência simples* ou *flat sequence* nesse contexto específico: como uma classificação para os tipos de sequência Python que não sejam containers. Na Wikipedia, esse uso seria rotulado de “original research” (pesquisa original). Prefiro chamá-lo de “nosso termo” na esperança de que você o ache útil e o adote também.

Listas com itens misturados

Os textos introdutórios de Python enfatizam o fato de que as listas podem conter objetos de tipos diferentes, porém, na prática, esse recurso não é muito útil: colocamos itens em uma lista para processá-los posteriormente, o que implica que todos os itens devem aceitar pelo menos algumas operações em comum (ou seja, todos eles devem fazer “quack”, sejam eles geneticamente 100% patos ou não). Por exemplo, não é possível ordenar uma lista em Python 3 a menos que os itens nela contidos sejam comparáveis:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
```

```
>>> sorted(l)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Ao contrário das listas, as tuplas geralmente armazenam itens de tipos diferentes. Isso é natural, considerando que cada item em uma tupla é realmente um campo e o tipo de cada campo é independente dos demais.

O argumento `key` é brilhante

O argumento opcional `key` de `list.sort`, `sorted`, `max` e `min` é uma ótima ideia. Outras linguagens obrigam você a fornecer uma função de comparação de dois argumentos, como a função obsoleta `cmp(a, b)` do Python 2. Usar `key` é mais simples e mais eficiente. É mais simples porque basta definir uma função de um argumento para obter ou calcular qualquer que seja o critério que você queira usar para ordenar os seus objetos; isso é mais fácil do que escrever uma função de dois argumentos para retornar `-1`, `0`, `1`. É também mais eficiente, pois a função `key` é chamada somente uma vez por item, enquanto a comparação de dois argumentos é chamada sempre que o algoritmo de ordenação precisar comparar dois itens. É claro que Python também precisa comparar as chaves enquanto estiver ordenando, porém essa comparação é feita em código C otimizado, e não em uma função Python escrita por você.

A propósito, usando `key`, conseguimos até ordenar uma mistura de números e strings que representem números. Basta decidir se você quer tratar todos os itens como inteiros ou como strings:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
>>> sorted(l, key=str)
[0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

Oracle, Google e a conspiração Timbot

O algoritmo de ordenação usado em `sorted` e em `list.sort` é o Timsort, um algoritmo adaptativo que alterna entre estratégias de `insertion sort` e `merge sort`, de acordo com o grau de ordenação dos dados. Ele é eficiente, pois dados do mundo real tendem a ter porções de itens ordenados. Há um artigo na Wikipedia (<http://en.wikipedia.org/wiki/Timsort>) sobre esse algoritmo.

O Timsort foi inicialmente implementado no CPython em 2002. Desde 2009, o Timsort também é usado para ordenar arrays tanto no Java padrão quanto no Android, um fato que se tornou amplamente conhecido quando a Oracle usou

parte do código relacionado ao Timsort como evidência de que o Google teria infringido a propriedade intelectual da Sun. Veja o site *Oracle v. Google – Day 14 Filings* (<http://bit.ly/1Vm6Ool>).

O Timsort foi criado por Tim Peters, um core developer de Python tão produtivo que acredita-se que ele seja uma IA, o Timbot. Você pode ler sobre essa teoria da conspiração em *Python Humor* (<https://www.python.org/doc/humor/#id9>). Tim também escreveu The Zen of Python: `import this`.

CAPÍTULO 3

Dicionários e conjuntos

Qualquer programa Python em execução tem diversos dicionários ativos simultaneamente, mesmo que o código do programa do usuário não utilize explicitamente um dicionário.

— A. M. Kuchling

Beautiful Code (O'Reilly, 2007), Capítulo 18, Python's Dictionary Implementation

O tipo `dict` não só é amplamente usado em nossos programas como também é parte fundamental da implementação de Python. Namespaces de módulos, atributos de classe e de instância e argumentos nomeados de funções são algumas das construções básicas em que os dicionários estão implementados. As próprias funções embutidas ficam em um dicionário: `__builtins__.__dict__`.

Por causa de seu papel essencial, os dicionários de Python são extremamente otimizados. *As tabelas hash são o mecanismo responsável pelo seu alto desempenho.*

Discutiremos também os conjuntos (`set`) neste capítulo, que também são implementados com tabelas hash. Saber como uma tabela hash funciona é fundamental para aproveitar ao máximo os dicionários e os conjuntos.

Eis uma síntese do que este capítulo contém:

- métodos comuns de dicionário;
- tratamento especial para chaves ausentes;
- variações de `dict` na biblioteca-padrão;
- os tipos `set` e `frozenset`;
- como as tabelas hash funcionam;
- implicações das tabelas hash (limitações dos tipos de chave, ordenação imprevisível etc.).

Tipos genéricos de mapeamento

O módulo `collections.abc` inclui as ABCs `Mapping` e `MutableMapping` para formalizar as interfaces de `dict` e de tipos semelhantes (em Python 2.6 a 3.2, essas classes são importadas do módulo `collections`, e não de `collections.abc`). Veja a figura 3.1.

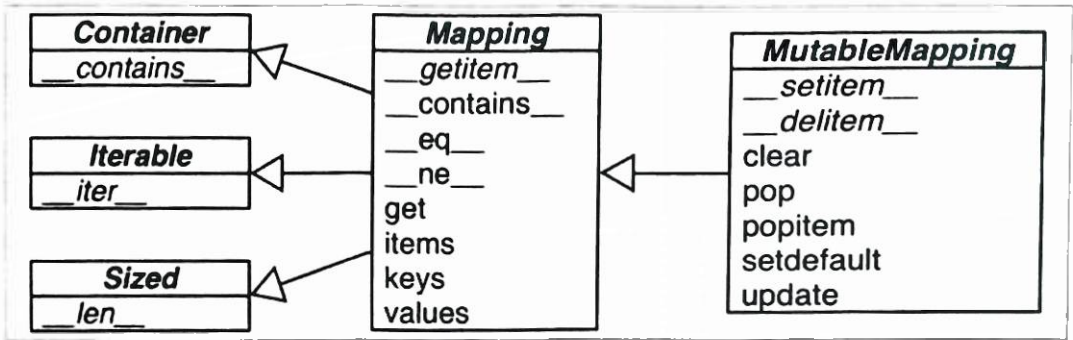


Figura 3.1 – Diagrama UML de classes para `MutableMapping` e suas superclasses de `collections.abc` (as setas de herança apontam das subclasses para as superclasses; os nomes em itálico são classes e métodos abstratos).

As implementações de mapeamentos (mappings) especializados geralmente estendem `dict` ou `collections.UserDict` em vez dessas ABCs. A importância fundamental das ABCs está em documentar e formalizar as interfaces mínimas para os mapeamentos, além de servir como critérios para testes com a função `isinstance` em códigos que devam oferecer suporte a mapeamentos de forma genérica:

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
```

Usar `isinstance` é melhor que verificar se o argumento de uma função é do tipo `dict`, pois, desse modo, tipos alternativos de mapeamento poderão ser utilizados.

Todos os tipos de mapeamento da biblioteca-padrão usam o tipo básico `dict` em sua implementação, portanto uma limitação comum a esses tipos é que as chaves precisam ser *hashable* (os valores não precisam ser *hashable*, somente as chaves).

O que é hashable?

Eis uma parte da definição de *hashable* extraída do *Python Glossary* (Glossário do Python, <http://bit.ly/1K4qjwE>):

Um objeto é *hashable* se tiver um valor de *hash* que nunca mude durante seu tempo de vida (deve ter um método `__hash__()`) e puder ser comparado com outros objetos (deve ter um método `__eq__()`). Objetos *hashable* comparados como iguais devem ter o mesmo valor de *hash*. [...]

Os tipos imutáveis atômicos (`str`, `bytes`, tipos numéricos) são todos hashable. Um `frozenset` é sempre hashable porque seus elementos devem ser hashable por definição. Um `tuple` será hashable somente se todos os seus itens forem hashable. Veja as tuplas `tt`, `tl` e `tf`:

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```



Quando este livro foi escrito, o *Python Glossary* (Glossário do Python, <http://bit.ly/1K4qjwE>) afirmava o seguinte: “Todos os objetos embutidos imutáveis de Python são hashable”, porém isso não está correto, pois um `tuple` é imutável, mas pode conter referências a objetos que não são hashable.

Os tipos definidos pelo usuário são hashable por padrão, pois seus valores de hash são seus `id()` e, na comparação, todos são diferentes. Se um objeto implementar um `__eq__` personalizado que leve em consideração o seu estado interno, ele poderá ser hashable somente se todos os seus atributos forem imutáveis.

Considerando essas regras básicas, podemos criar dicionários de várias maneiras. A página *Built-in Types* (Tipos embutidos, <http://bit.ly/1QS9Ong>) da *Library Reference* (Guia de referência da biblioteca) apresenta o exemplo a seguir para mostrar as diversas formas de criar um dicionário:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Além da sintaxe de literais e do construtor `dict` flexível, podemos usar as *dict comprehensions* para criar dicionários. Veja a seção a seguir.

dict comprehensions

A partir de Python 2.7, a sintaxe de listcomps e de genexps passou a ser aplicada a dict comprehensions (e também às set comprehensions, que veremos em breve). Uma *dictcomp* cria uma instância de dict produzindo pares key:value a partir de qualquer iterável. O exemplo 3.1 mostra o uso de dict comprehensions para criar dois dicionários a partir da mesma lista de tuplas.

Exemplo 3.1 – Exemplos de dict comprehensions

```
>>> DIAL_CODES = [                                ❶
...     (86, 'China'),
...     (91, 'India'),
...     (1, 'United States'),
...     (62, 'Indonesia'),
...     (55, 'Brazil'),
...     (92, 'Pakistan'),
...     (880, 'Bangladesh'),
...     (234, 'Nigeria'),
...     (7, 'Russia'),
...     (81, 'Japan'),
... ]
>>> country_code = {country: code for code, country in DIAL_CODES} ❷
>>> country_code
{'China': 86, 'India': 91, 'Bangladesh': 880, 'United States': 1,
 'Pakistan': 92, 'Japan': 81, 'Russia': 7, 'Brazil': 55, 'Nigeria':
 234, 'Indonesia': 62}
>>> {code: country.upper() for country, code in country_code.items()} ❸
... if code < 66}
{1: 'UNITED STATES', 55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA'}
```

- ❶ Uma lista de pares pode ser usada diretamente com o construtor de dict.
- ❷ Nesse caso, os pares são invertidos: country é a chave e code é o valor.
- ❸ Invertendo os pares novamente, com os valores em letras maiúsculas e os itens filtrados com code < 66.

Se você está acostumado a usar liscomps, as dictcomps são um próximo passo natural. Se não estiver, esses novos usos da sintaxe de listcomp mostram que, mais do que nunca, é vantajoso tornar-se fluente nela.

Vamos agora apresentar uma visão panorâmica da API para mapeamentos.

Visão geral dos métodos comuns a mapeamentos

A API básica para mapeamentos é bem rica. A tabela 3.1 mostra os métodos implementados por dict e duas de suas variações mais úteis: defaultdict e OrderedDict, ambas definidas no módulo collections.

Tabela 3.1 – Métodos dos tipos de mapeamento dict, collections.defaultdict e collections.OrderedDict (métodos comuns de object foram omitidos por questões de concisão); argumentos opcionais estão entre [...]

	dict	defaultdict	OrderedDict	
d.clear()	●	●	●	Remove todos os itens
d.__contains__(k)	●	●	●	k in d
d.copy()	●	●	●	Cópia rasa
d.__copy__()		●		Suporte para copy.copy
d.default_factory		●		Função a ser chamada por __missing__ para gerar valores ausentes ^a
d.__delitem__(k)	●	●	●	del d[k] – remove item com a chave k
d.fromkeys(it, [initial])	●	●	●	Novo mapeamento a partir das chaves do iterável, com valor inicial opcional (default é None)
d.get(k, [default])	●	●	●	Obtém item com a chave k; devolve default ou None se estiver ausente
d.__getitem__(k)	●	●	●	d[k] – obtém item com a chave k
d.items()	●	●	●	Obtém view sobre itens – pares (key, value)
d.__iter__()	●	●	●	Obtém um iterador para chaves
d.keys()	●	●	●	Obtém view para chaves
d.__len__()	●	●	●	len(s) – número de itens
d.__missing__(k)		●		Chamado quando __getitem__ não encontra a chave
d.move_to_end(k, [last])			●	Move k para a primeira ou para a última posição (last é True por default)
d.pop(k, [default])	●	●	●	Remove e devolve o valor em k, ou default ou None se estiver ausente
d.popitem()	●	●	●	Remove e devolve um item (key, value) arbitrário ^b
d.__reversed__()			●	Obtém um iterador para chaves do último para o primeiro inserido
d.setdefault(k, [default])	●	●	●	Se k in d, devolve d[k]; caso contrário, define d[k] = default e devolve esse valor
d.__setitem__(k, v)	●	●	●	d[k] = v – coloca v em k

	dict	defaultdict	OrderedDict	
d.update(m, **kargs)	•	•	•	Atualiza d com itens do mapeamento ou do iterável de pares (key, value)
d.values()	•	•	•	Obtém <i>view</i> dos valores

- `default_factory` não é um método, mas um atributo de instância invocável (callable) definido pelo usuário final quando `defaultdict` é instanciado.
- `OrderedDict.popitem()` remove o primeiro item inserido (FIFO); um argumento `last` opcional, se definido com `True`, remove o último item (LIFO).

O modo como `update` trata seu primeiro argumento `m` é um ótimo exemplo de *duck typing* (tipagem pato): inicialmente, ele verifica se `m` tem um método `keys` e, em caso afirmativo, supõe que é um mapeamento. Caso contrário, `update` faz a iteração por `m`, supondo que seu itens sejam pares (key, value). O construtor da maioria dos mapeamentos em Python usa a lógica de `update` internamente, o que significa que eles podem ser inicializados a partir de outros mapeamentos ou de qualquer objeto iterável que gere pares (key, value).

Um método peculiar de mapeamento é `setdefault`. Nem sempre precisamos dele, mas, quando precisamos, esse método ganha tempo evitando buscas redundantes. Se você não se sente à vontade para usá-lo, a seção a seguir explicará como fazer isso com um exemplo prático.

Tratando chaves ausentes com `setdefault`

De acordo com a filosofia de *falhar rapidamente* (*fail fast*), acessos a `dict` com `d[k]` geram erro quando `k` for uma chave inexistente. Todo Pythonista sabe que `d.get(k, default)` é uma alternativa a `d[k]` sempre que um valor `default` for mais conveniente que tratar `KeyError`. No entanto, ao atualizar o valor encontrado (se for mutável), usar `__getitem__` ou `get` não é prático nem eficiente. Considere o exemplo 3.2, que apresenta um script não otimizado, criado somente para mostrar um caso em que `dict.get` não é a maneira mais adequada de tratar uma chave ausente.

O exemplo 3.2 foi adaptado de um exemplo de Alex Martelli¹ que gera um índice, como mostra o exemplo 3.3.

Exemplo 3.2 – `index0.py` utiliza `dict.get` para buscar e atualizar uma lista de ocorrências de palavras a partir do índice (uma solução melhor será apresentada no exemplo 3.4)

```
"""Cria um índice que mapeia palavra -> lista de ocorrências"""

import sys
```

¹ O script original aparece no slide 41 da apresentação “Re-learning Python” (Reaprendendo Python, <http://bit.ly/1QmmPFf>) de Martelli. Seu script, na verdade, é uma demonstração de `dict.setdefault`, como mostra o nosso exemplo 3.4.

```

import re
WORD_RE = re.compile('\w+')
index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            # isto não é elegante; foi codificado desta forma para ilustrar uma questão
            occurrences = index.get(word, []) ❶
            occurrences.append(location)      ❷
            index[word] = occurrences        ❸

# exibe em ordem alfabética
for word in sorted(index, key=str.upper): ❹
    print(word, index[word])

```

- ❶ Obtém a lista de ocorrências para word, ou [] se essa palavra não for encontrada.
- ❷ Concatena a nova posição a occurrences.
- ❸ Coloca occurrences alterado no dicionário index; isso implica uma segunda busca em index.
- ❹ No argumento key= de sorted, não estou chamando str.upper; estou apenas passando uma referência a esse método para que a função sorted possa usá-lo a fim de normalizar as palavras para a ordenação.²

Exemplo 3.3 – Saída parcial do exemplo 3.2 processando o Zen of Python; cada linha mostra uma palavra e uma lista de ocorrências representadas como pares: (número-da-linha, número-da-coluna)

```

$ python3 index0.py ../../data/zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
Beautiful [(3, 1)]

```

² Esse é um exemplo de uso de um método como função de primeira classe, que será o assunto do capítulo 5.

```
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11),
(17, 8), (18, 25)]
```

```
***
```

As três linhas que lidam com occurrences no exemplo 3.2 podem ser substituídas por uma única linha usando `dict.setdefault`. O exemplo 3.4 é mais parecido com o exemplo original de Alex Martelli.

Exemplo 3.4 – `index.py` utiliza `dict.setdefault` para buscar e atualizar uma lista de ocorrências de palavras do índice em uma única linha; compare com o exemplo 3.2

```
"""Cria um índice que mapeia palavra -> lista de ocorrências"""
```

```
import sys
import re

WORD_RE = re.compile('\w+')

index = {}

with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location) ❶

# exibe em ordem alfabética
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

- ❶ Obtém a lista de ocorrências para `word` ou define-a com `[]` se não for encontrada; `setdefault` devolve o valor, portanto poderá ser atualizado sem exigir uma segunda busca.

Em outras palavras, o resultado final desta linha:

```
my_dict.setdefault(key, []).append(new_value)
```

... será igual à execução de:

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

... exceto pelo fato de o último código executar pelo menos duas buscas de chave – três se ela não for encontrada – enquanto `setdefault` faz tudo com uma única busca.

Um problema relacionado a esse que acabamos de ver, ou seja, o tratamento de chaves ausentes em qualquer consulta (e não apenas na inserção), será o assunto da próxima seção

Mapeamentos com consulta de chave flexível

Às vezes, é conveniente ter mapeamentos que devolvam alguns valores predefinidos quando uma chave ausente é buscada. Há duas abordagens principais nesse caso: uma delas é usar um `defaultdict` no lugar de um simples `dict`; a outra opção é usar uma subclasse de `dict` ou de qualquer outro tipo de mapeamento e acrescentar um método `__missing__`. Ambas as soluções serão discutidas a seguir.

`defaultdict`: outra abordagem para chaves ausentes

O exemplo 3.5 usa `collections.defaultdict` para oferecer outra solução elegante ao problema do exemplo 3.4. Um `defaultdict` é configurado para criar itens sob demanda sempre que uma chave ausente for buscada.

Eis como isso funciona: ao instanciar um `defaultdict`, você fornece uma função ou classe que será usada para gerar um valor default sempre que `__getitem__` receber um argumento de chave inexistente.

Por exemplo, dado um `defaultdict` vazio criado com `dd = defaultdict(list)`, se `'new-key'` não estiver em `dd`, a expressão `dd['new-key']` executará os seguintes passos:

1. Chama `list()` para criar uma nova lista.
2. Insere a lista em `dd` usando `'new-key'` como chave.
3. Devolve uma referência a essa lista.

O objeto invocável (callable) que gera os valores default é armazenado em um atributo de instância chamado `default_factory`.

Exemplo 3.5 – `index_default.py`: usando uma instância de `defaultdict` no lugar do método `setdefault`

```
"""Cria um índice que mapeia palavra -> lista de ocorrências"""
```

```
import sys
import re
import collections

WORD_RE = re.compile('\w+')

index = collections.defaultdict(list) ❶

with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
```

```

        location = (line_no, column_no)
        index[word].append(location) ❷

# exibe em ordem alfabética
for word in sorted(index, key=str.upper):
    print(word, index[word])

```

❶ Cria um `defaultdict` com o construtor `list` como `default_factory`.

❷ Se `word` não estiver inicialmente em `index`, `default_factory` será chamado para gerar o valor ausente, que, nesse caso, é uma `list` vazia; ela será então atribuída a `index[word]` e devolvida, de modo que a operação `.append(location)` sempre será bem-sucedida.

Se `default_factory` não for especificado, `KeyError` será gerado como sempre para as chaves ausentes.



O `default_factory` de um `defaultdict` será chamado somente para fornecer valores default para chamadas a `__getitem__`, e não para outros métodos. Por exemplo, se `dd` for um `defaultdict` e `k` for uma chave ausente, `dd[k]` chamará `default_factory` para criar um valor default, mas `dd.get(k)` continuará devolvendo `None`.

O mecanismo que faz `defaultdict` funcionar chamando `default_factory`, na verdade, é o método especial `__missing__` – um recurso aceito por todos os tipos-padrões de mapeamento, que será discutido a seguir.

Método `__missing__`

Subjacente ao modo como os mapeamentos lidam com chaves ausentes está o método `__missing__`, cujo nome é bastante apropriado. Esse método não está definido na classe base `dict`, porém `dict` está ciente de sua finalidade: se você criar uma subclasse de `dict` e implementar um método `__missing__`, o método `dict.__getitem__` padrão o chamará sempre que uma chave não for encontrada, em vez de gerar um `KeyError`.



O método `__missing__` é chamado apenas por `__getitem__` (isto é, para o operador `d[k]`). A presença de um método `__missing__` não terá nenhum efeito no comportamento de outros métodos que consultem chaves, por exemplo, `get` ou `__contains__` (que implementa o operador `in`). É por isso que o `default_factory` de `defaultdict` funciona somente com `__getitem__`, conforme observado no aviso no final da seção anterior.

Suponha que você queira um mapeamento em que as chaves sejam convertidas para `str` quando consultadas. Um caso de uso concreto é o projeto `Pingo.io` (<http://www.pingo.io/docs/>), em que uma placa programável com pinos GPIO (por

exemplo, Raspberry Pi ou Arduino) é representada por um objeto `board` com um atributo `board.pins`; esse atributo contém um mapeamento das localizações físicas dos pinos para objetos que representam os pinos. A localização física pode ser apenas um número ou uma string como "A0" ou "P9_12". Por questões de consistência, é desejável que todas as chaves em `board.pins` sejam strings, mas é conveniente que a consulta a `my_arduino.pin[13]` também funcione de modo que programadores iniciantes não tenham problemas quando quiserem fazer piscar o LED no pino 13 de seus Arduinos. O exemplo 3.6 mostra como um mapeamento desse tipo funcionaria.

Exemplo 3.6 – Ao buscar uma chave que não seja uma string, `StrKeyDict0` a converterá para `str` se ela não for encontrada

Tests for item retrieval using ``d[key]`` notation::

```
>>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
>>> d['2']
'two'
>>> d[4]
'four'
>>> d[1]
Traceback (most recent call last):
...
KeyError: '1'
```

Tests for item retrieval using ``d.get(key)`` notation::

```
>>> d.get('2')
'two'
>>> d.get(4)
'four'
>>> d.get(1, 'N/A')
'N/A'
```

Tests for the ``in`` operator::

```
>>> 2 in d
True
>>> 1 in d
False
```

O exemplo 3.7 implementa uma classe `StrKeyDict0` que faz os testes anteriores passarem.



Uma maneira melhor de criar um tipo de mapeamento definido pelo usuário é criar uma subclasse de `collections.UserDict` no lugar de `dict` (como faremos no exemplo 3.8). Nesse caso, criaremos uma subclasse de `dict` somente para mostrar que `__missing__` é invocado pelo método embutido `dict.__getitem__`.

Exemplo 3.7 – StrKeyDict0 converte chaves que não sejam string para str na consulta (veja os testes no exemplo 3.6)

```
class StrKeyDict0(dict): ❶
    def __missing__(self, key):
        if isinstance(key, str): ❷
            raise KeyError(key)
            return self[str(key)] ❸
        def get(self, key, default=None):
            try:
                return self[key] ❹
            except KeyError:
                return default ❺
        def __contains__(self, key):
            return key in self.keys() or str(key) in self.keys() ❻
```

- ❶ StrKeyDict0 herda de dict.
- ❷ Verifica se key já é uma str. Se for e estiver ausente, gera KeyError.
- ❸ Cria str a partir de key e refaz a consulta.
- ❹ O método get delega para `__getitem__` usando a notação `self[key]`; isso dá a oportunidade ao nosso `__missing__` de agir.
- ❺ Se um `KeyError` for gerado, é sinal de que `__missing__` já falhou, portanto devolvemos `default`.
- ❻ Procura a chave não modificada (a instância pode conter chaves que não sejam str) e, em seguida, procura uma str criada a partir da chave.

Reserve tempo para considerar por que o teste `isinstance(key, str)` é necessário na implementação de `__missing__`.

Sem esse teste, nosso método `__missing__` funcionaria sem problemas para qualquer chave `k` – seja str ou não – sempre que `str(k)` gerasse uma chave existente. Entretanto, se `str(k)` não for uma chave existente, teremos uma recursão infinita. Na última linha, `self[str(key)]` chamaria `__getitem__` passando essa chave str que, por sua vez, chamaria `__missing__` novamente.

O método `__contains__` também é necessário para um comportamento consistente nesse exemplo, pois a operação `k in d` o chama, porém o método herdado de `dict` não chama `__missing__` como alternativa. Há um detalhe sutil em nossa implementação de `__contains__`: não verificamos a chave da maneira pythônica usual – `k in my_dict` – porque `str(key) in self` chamaria `__contains__` recursivamente. Evitamos isso consultando a chave em `self.keys()` explicitamente.



Uma busca como `k in my_dict.keys()` é eficiente em Python 3 mesmo para mapeamentos bem grandes porque `dict.keys()` devolve uma view semelhante a um conjunto, e as verificações de existência em conjuntos são tão rápidas quanto em dicionários. Os detalhes estão documentados na seção “Dictionary view objects” (Objetos view de dicionários) da documentação (<http://bit.ly/1Vm7E4q>). Em Python 2, `dict.keys()` devolve uma `list`, portanto nossa solução também funciona nessa versão, porém não é eficiente para dicionários grandes, pois `k in my_list` precisa percorrer a lista.

A verificação para a chave não modificada – `key in self.keys()` – é necessária para que o código esteja correto, pois `StrKeyDict0` não obriga que todas as chaves do dicionário devam ser do tipo `str`. Nosso objetivo com esse exemplo simples não é oferecer garantias para os tipos, mas apenas deixar a busca “mais amigável”.

Até agora, discutimos os tipos de mapeamento `dict` e `defaultdict`, mas a biblioteca-padrão vem com outras implementações de mapeamento que serão discutidas a seguir.

Variações de dict

Nesta seção, resumiremos os diversos tipos de mapeamento incluídos no módulo `collections` da biblioteca-padrão, além de `defaultdict`:

`collections.OrderedDict`

Mantém as chaves na ordem de inserção, permitindo que a iteração pelos itens seja feita em uma ordem previsível. O método `popitem` de um `OrderedDict` remove o primeiro item por padrão, porém, se chamado com `my_odict.popitem(last=True)`, o último item adicionado será removido.

`collections.ChainMap`

Armazena uma lista de mapeamentos que podem ser buscados como se fossem um só. A consulta é feita em cada mapeamento na sequência e será bem-sucedida se a chave for encontrada em qualquer um deles. Isso é útil para interpretadores de linguagens com escopos aninhados, em que cada mapeamento representa o contexto de um escopo. A seção “ChainMap objects” (Objetos ChainMap) da documentação de `collections` (<http://bit.ly/1Vm7I4c>) tem diversos exemplos do uso de `ChainMap`, incluindo o trecho de código a seguir, inspirado nas regras básicas de consulta a variáveis em Python:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

`collections.Counter`

Um mapeamento que armazena um contador inteiro para cada chave. Atualizar uma chave existente faz o contador ser incrementado. Isso pode ser usado para contar instâncias de objetos hashable (as chaves) ou como um multiset – um conjunto capaz de armazenar diversas ocorrências de cada elemento. `Counter` implementa os operadores `+` e `-` para combinar contadores, além de outros métodos úteis como `most_common([n])`, que devolve uma lista ordenada de tuplas com os *n* itens mais comuns e seus contadores; consulte a documentação (<http://bit.ly/1JHVi2E>). No exemplo a seguir, `Counter` é usado para contar letras em palavras:

```
>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.update('aaaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(2)
[('a', 10), ('z', 3)]
```

`collections.UserDict`

É uma implementação Python pura de um mapeamento que funciona como um `dict` padrão.

Enquanto `OrderedDict`, `ChainMap` e `Counter` estão prontos para uso, `UserDict` foi concebido para que subclasses sejam criadas a partir dele, como faremos a seguir.

Criando subclasses de `UserDict`

Quase sempre, é mais fácil criar um novo tipo de mapeamento estendendo `UserDict` em vez de `dict`. O valor de `UserDict` pode ser apreciado ao estendermos o `StrKeyDict0` do exemplo 3.7 para garantir que qualquer chave adicionada ao mapeamento seja armazenada como `str`.

O principal motivo pelo qual é preferível criar uma subclasse de `UserDict` em vez de `dict` é que o tipo embutido tem alguns atalhos de implementação que acabam nos forçando a sobrescrever métodos que podemos simplesmente herdar de `UserDict` sem nenhum problema.³

Observe que `UserDict` não herda de `dict`, porém tem uma instância de `dict` interna chamada `data`, que armazena os itens propriamente ditos. Isso evita uma recursão

³ O problema exato com a criação de subclasses de `dict` e de outros tipos embutidos será discutido na seção “Artimanhas da criação de subclasses de tipos embutidos” na página 396.

indesejada na codificação de métodos especiais como `__setitem__` e simplifica a implementação de `__contains__` se comparada ao exemplo 3.7.

Graças a `UserDict`, `StrKeyDict` (Exemplo 3.8), de fato, é menor que `StrKeyDict0` (Exemplo 3.7), mas faz mais: `StrKeyDict` armazena todas as chaves como `str`, evitando surpresas desagradáveis se a instância for criada ou atualizada com dados contendo chaves que não sejam strings.

Exemplo 3.8 – `StrKeyDict` sempre converte chaves que não são strings para `str` – na inserção, na atualização e na consulta

```
import collections

class StrKeyDict(collections.UserDict): ❶
    def __missing__(self, key): ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]
    def __contains__(self, key):
        return str(key) in self.data ❸
    def __setitem__(self, key, item):
        self.data[str(key)] = item ❹
```

- ❶ `StrKeyDict` estende `UserDict`.
- ❷ `__missing__` é exatamente como no exemplo 3.7.
- ❸ `__contains__` é mais simples: podemos supor que todas as chaves armazenadas sejam `str` e consultar `self.data` em vez de chamar `self.keys()` como fizemos em `StrKeyDict0`.
- ❹ `__setitem__` converte qualquer `key` para uma `str`. Esse método é mais fácil de sobrescrever quando podemos delegar ao atributo `self.data`.

Como `UserDict` é uma subclasse de `MutableMapping`, os métodos restantes que tornam `StrKeyDict` um mapeamento completo são herdados de `UserDict`, `MutableMapping` ou `Mapping`. Os últimos têm diversos métodos concretos úteis, apesar de serem ABCs (abstract base classes, ou classes-base abstratas). Vale a pena dar uma olhada nos métodos a seguir:

`MutableMapping.update`

Esse método poderoso pode ser chamado diretamente, mas é usado também por `__init__` para carregar a instância a partir de outros mapeamentos, iteráveis de pares (`key`, `value`) e argumentos nomeados. Por usar `self[key] = value` para adicionar itens, nossa implementação de `__setitem__` é chamada.

Mapping.get

Em `StrKeyDict0` (Exemplo 3.7), tivemos que codificar nosso próprio `get` para obter resultados consistentes com `__getitem__`, porém, no exemplo 3.8, herdamos `Mapping.get`, que é implementado exatamente como `StrKeyDict0.get` (veja o código-fonte Python em <http://bit.ly/1FEOPPB>).



Depois que escrevi `StrKeyDict`, descobri que Antoine Pitrou redigiu a *PEP 455 – Adding a key-transforming dictionary to collections* (Adicionando um dicionário transformador de chaves às coleções, <https://www.python.org/dev/peps/pep-0455/>) e um patch para aperfeiçoar o módulo `collections` com um `TransformDict`. O patch está associado ao *issue18986* (<http://bugs.python.org/issue18986>) e poderá ser incluído em Python 3.5. Para testar, extraí `TransformDict` em um módulo independente [*03-dict-set/transformdict.py* (<http://bit.ly/1Vm7OJ5>) no repositório de código de *Python fluente* (<https://github.com/fluentpython/example-code>)]. `TransformDict` é mais genérico que `StrKeyDict` e mais complexo por causa do requisito de preservar as chaves como foram originalmente inseridas.

Sabemos que há vários tipos de sequências imutáveis, mas e um mapeamento imutável? Bem, na verdade, não há nenhum na biblioteca-padrão, porém um substituto está disponível. Continue lendo.

Mapeamentos imutáveis

Os tipos de mapeamento oferecidos pela biblioteca-padrão são todos imutáveis; às vezes pode ser desejável garantir que um usuário não altere um mapeamento por engano. Um caso de uso concreto pode ser visto, novamente, no projeto *Pingo.io* que descrevi na seção “Método `__missing__`” na página 102: o mapeamento `board.pins` representa os pinos GPIO físicos do dispositivo. Sendo assim, seria bom evitar atualizações feitas inadvertidamente em `board.pins`, pois o hardware não pode ser alterado por software; portanto qualquer alteração no mapeamento o deixará inconsistente com a realidade física do dispositivo.

A partir de Python 3.3, o módulo `types` oferece uma classe wrapper chamada `MappingProxyType`; dado um mapeamento, essa classe devolve uma instância de `mappingproxy`, que é uma view somente de leitura, porém dinâmica, do mapeamento original. Isso quer dizer que as atualizações no mapeamento original podem ser vistas em `mappingproxy`, mas não será possível fazer alterações por meio dela. Veja uma rápida demonstração no exemplo 3.9.

Exemplo 3.9 – MappingProxyType cria uma instância de mappingproxy somente para leitura a partir de um dict

```
>>> from types import MappingProxyType
>>> d = {1: 'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1] ❶
'A'
>>> d_proxy[2] = 'x' ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy ❸
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

- ❶ Itens em `d` podem ser vistos por meio de `d_proxy`.
- ❷ Alterações não podem ser feitas por meio de `d_proxy`.
- ❸ `d_proxy` é dinâmico: qualquer alteração em `d` se refletirá em `d_proxy`.

Eis como isso poderia ser usado na prática no cenário de Pingo.io: o construtor em uma subclasse concreta de `Board` preencheria um mapeamento privado com os objetos que representam os pinos e o exporia aos clientes da API por meio de um atributo `.pins` público implementado como um `mappingproxy`. Dessa maneira, os clientes não poderiam adicionar, remover ou alterar os pinos acidentalmente.⁴

Agora que já discutimos a maioria dos tipos de mapeamento da biblioteca-padrão e quando usá-los, passaremos para a discussão sobre conjuntos.

Teoria dos conjuntos

Os conjuntos (sets) são um acréscimo relativamente novo na história de Python e, de certo modo, são subutilizados. O tipo `set` e seu irmão imutável `frozenset` apareceram inicialmente em um módulo de Python 2.3 e foram promovidos a tipos embutidos em Python 2.6.

⁴ Na verdade, não usamos `MappingProxyType` em Pingo.io porque ele é uma novidade acrescentada em Python 3.3 e, no momento, precisamos oferecer suporte a Python 2.7.



Neste livro, a palavra “conjunto” é usada para referir-se tanto a `set` quanto a `frozenset`. Ao falarmos especificamente da classe `set`, seu nome aparecerá na fonte de largura constante usada para códigos-fonte: `set`.

Um conjunto é uma coleção de objetos únicos. Um caso de uso básico é remover duplicações:

```
>>> l = ['spam', 'spam', 'eggs', 'spam']
>>> set(l)
{'eggs', 'spam'}
>>> list(set(l))
['eggs', 'spam']
```

Elementos de conjuntos devem ser hashable. O tipo `set` não é hashable, mas `frozenset` é, portanto podemos ter elementos `frozenset` em um `set`.

Além de garantir a unicidade, os conjuntos implementam as operações essenciais de conjuntos como operadores infixos; sendo assim, dados dois conjuntos `a` e `b`, `a | b` devolve a união, `a & b` calcula a intersecção e `a - b` calcula a diferença entre os conjuntos. O uso inteligente das operações de conjuntos pode reduzir tanto o número de linhas quanto o tempo de execução de programas Python, ao mesmo tempo, deixando o código mais fácil de ler e de compreender – devido à remoção de laços e de muita lógica condicional.

Por exemplo, suponha que você tenha um conjunto grande de endereços de email (`haystack`) e tenha um conjunto menor de endereços (`needles`) e deva contar quantos `needles` existem em `haystack`⁵. Graças à intersecção (operador `&`), podemos codificar isso em uma única linha (veja o exemplo 3.10).

Exemplo 3.10 – Conta ocorrências de `needles` em `haystack`, ambos do tipo `set`

```
found = len(needles & haystack)
```

Sem o operador de intersecção, você escreveria o exemplo 3.11 para fazer a mesma tarefa do exemplo 3.10.

Exemplo 3.11 – Conta ocorrências de `needles` em `haystack` (mesmo resultado final do exemplo 3.10)

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

⁵ N.T.: A lógica dos identificadores “haystack” e “needle” usada neste exemplo e em outros que envolvem buscas: “haystack” é palheiro, e “needle” é agulha, em referência à expressão “procurar uma agulha em um palheiro”.

O exemplo 3.10 é um pouco mais rápido que o exemplo 3.11. Por outro lado, o exemplo 3.11 funciona para quaisquer objetos iteráveis *needles* e *haystack*, enquanto o exemplo 3.10 exige que ambos sejam conjuntos. No entanto, se não tiver conjuntos à mão, você sempre poderá criá-los durante a execução, conforme mostra o exemplo 3.12.

Exemplo 3.12 – Conta ocorrências de *needles* em *haystack*; estas linhas funcionam para qualquer tipo iterável

```
found = len(set(needles) & set(haystack))  
  
# outra maneira:  
found = len(set(needles).intersection(haystack))
```

É claro que há um custo extra envolvido na criação dos conjuntos no exemplo 3.12, mas, se *needles* ou *haystack* já forem conjuntos, as alternativas no exemplo 3.12 poderão ser menos custosas que o exemplo 3.11.

Qualquer um dos exemplos anteriores é capaz de buscar mil valores em um *haystack* de dez milhões de itens em pouco mais de três milissegundos – isso corresponde a aproximadamente três microssegundos por item.

Além de verificar muito rápido se um elemento está presente ou não (graças à tabela hash subjacente), os tipos embutidos *set* e *frozenset* oferecem uma enorme variedade de operações para criar novos conjuntos ou, no caso de *set*, alterar conjuntos existentes. Discutiremos as operações em breve, mas, antes disso, farei uma observação sobre a sintaxe.

Literais de set

A sintaxe literal de *set* – `{1}`, `{1, 2}` etc. – se parece exatamente com a notação matemática, com uma exceção importante: não existe uma notação literal para o *set* vazio, portanto devemos nos lembrar de escrever `set()`.



Idiosincrasia da sintaxe

Não se esqueça: para criar um *set* vazio, utilize o construtor sem argumentos – `set()`. Se escrever `{}`, você criará um *dict* vazio – isso não mudou.

Em Python 3, a representação-padrão em string dos conjuntos sempre usa a notação `{...}`, exceto para o conjunto vazio:

```
>>> s = {1}  
>>> type(s)  
<class 'set'>  
>>> s  
{1}  
>>> s.pop()
```

```
1
>>> s
set()
```

A sintaxe de um set literal como `{1, 2, 3}` é mais rápida e mais legível que chamar o construtor (por exemplo, `set([1, 2, 3])`). Esta última forma é mais lenta porque, para avaliá-la, o interpretador Python precisa pesquisar o nome `set` para encontrar o construtor, então construir uma lista e, por fim, passá-la ao construtor. Em comparação, para processar um literal como `{1, 2, 3}`, o interpretador Python executa um bytecode especializado: `BUILD_SET`.

Dê uma olhada no bytecode das duas operações, conforme exibidos por `dis.dis` (a função disassembler):

```
>>> from dis import dis
>>> dis('{1}')
1          0 LOAD_CONST          0 (1)
           3 BUILD_SET           1
           6 RETURN_VALUE

>>> dis('set([1])')
1          0 LOAD_NAME           0 (set)
           3 LOAD_CONST          0 (1)
           6 BUILD_LIST         1
           9 CALL_FUNCTION       1 (1 positional, 0 keyword pair)
          12 RETURN_VALUE
```

- ❶ Gera o bytecode para a expressão literal `{1}`.
 - ❷ O bytecode especial `BUILD_SET` faz quase todo o trabalho.
 - ❸ Bytecode para `set([1])`.
 - ❹ Três operações no lugar de `BUILD_SET`: `LOAD_NAME`, `BUILD_LIST` e `CALL_FUNCTION`.
-

Não existe uma sintaxe especial para representar literais `frozenset` – eles precisam ser criados por meio da chamada ao construtor. Em Python 3, a representação-padrão em string se parece com uma chamada ao construtor `frozenset`. Observe a saída na sessão do console:

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

Falando em sintaxe, a forma familiar das listcomps foi adaptada para a criação de conjuntos também.

Set comprehensions

As set comprehensions (*setcomps*) foram adicionadas em Python 2.7, juntamente com as dictcomps que vimos na seção “Dict comprehensions” na página 96. O exemplo 3.13 mostra um caso simples.

Exemplo 3.13 – Cria um conjunto de caracteres Latin-1 que tenham a palavra “SIGN” em seus nomes em Unicode

```
>>> from unicodedata import name ❶
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')} ❷
{'$', '=', 'ç', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', 'ø',
'°', '+', '÷', '±', '>', '¬', '®', '%'}
```

- ❶ Importa a função `name` de `unicodedata` para obter os nomes dos caracteres.
- ❷ Cria um conjunto de caracteres com os códigos de 32 a 255 que tenham a palavra `'SIGN'` em seus nomes.

Deixando de lado a questão da sintaxe, vamos agora analisar a enorme variedade de operações oferecida pelos conjuntos.

Operações de conjuntos

A figura 3.2 apresenta uma visão geral dos métodos que podemos esperar de conjuntos mutáveis e imutáveis. Muitos deles são métodos especiais para sobrecarga de operador. A tabela 3.2 mostra os operadores matemáticos de conjuntos com operadores ou métodos correspondentes em Python. Observe que alguns operadores e métodos realizam alterações in-place no conjunto-alvo (por exemplo, `&=`, `difference_update` etc.). Essas operações não fazem sentido no mundo ideal dos conjuntos matemáticos e não estão implementadas em `frozenset`.

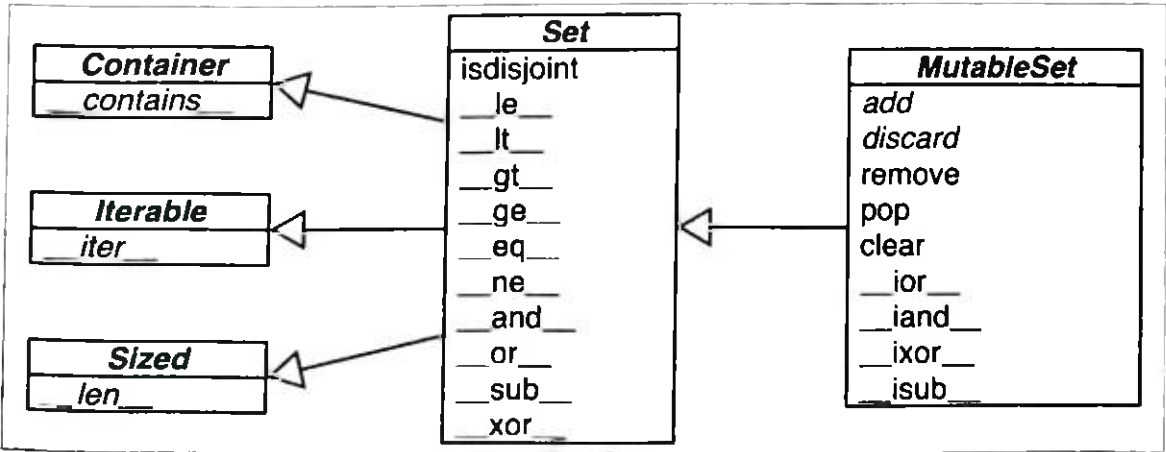


Figura 3.2 – Diagrama de classes UML para `MutableSet` e suas superclasses de `collections.abc` (os nomes em *itálico* são classes e métodos abstratos; os métodos de operadores reversos foram omitidos por questões de concisão).



Os operadores infixos na tabela 3.2 exigem que ambos os operandos sejam conjuntos, porém todos os demais métodos aceitam um ou mais argumentos iteráveis. Por exemplo, para calcular a união de quatro coleções *a*, *b*, *c* e *d*, podemos chamar *a.union(b, c, d)*, em que *a* deve ser um set, porém *b*, *c* e *d* podem ser iteráveis de qualquer tipo.

Tabela 3.2 – Operações matemáticas de set: esses métodos geram um novo conjunto ou atualizam o conjunto-alvo in-place se ele for mutável

Símbolo matemático	Operador Python	Método	Descrição
$s \cap z$	<code>s & z</code>	<code>s.__and__(z)</code>	Intersecção entre <i>s</i> e <i>z</i>
	<code>z & s</code>	<code>s.__rand__(z)</code>	Operador & reverso
		<code>s.intersection(it, ...)</code>	Intersecção entre <i>s</i> e todos os conjuntos criados a partir dos iteráveis <i>it</i> etc.
	<code>s &= z</code>	<code>s.__iand__(z)</code>	<i>s</i> atualizado com a intersecção entre <i>s</i> e <i>z</i>
		<code>s.intersection_update(it, ...)</code>	<i>s</i> atualizado com a intersecção entre <i>s</i> e todos os conjuntos criados a partir dos iteráveis <i>it</i> etc.
$s \cup z$	<code>s z</code>	<code>s.__or__(z)</code>	União de <i>s</i> e <i>z</i>
	<code>z s</code>	<code>s.__ror__(z)</code>	reverso
		<code>s.union(it, ...)</code>	União de <i>s</i> e todos os conjuntos criados a partir dos iteráveis <i>it</i> etc.
	<code>s = z</code>	<code>s.__ior__(z)</code>	<i>s</i> atualizado com a união de <i>s</i> e <i>z</i>
		<code>s.update(it, ...)</code>	<i>s</i> atualizado com a união de <i>s</i> e todos os conjuntos criados a partir dos iteráveis <i>it</i> etc.
$s \setminus z$	<code>s - z</code>	<code>s.__sub__(z)</code>	Complemento relativo ou diferença entre <i>s</i> e <i>z</i>
	<code>z - s</code>	<code>s.__rsub__(z)</code>	Operador - reverso
		<code>s.difference(it, ...)</code>	Diferença entre <i>s</i> e todos os conjuntos criados a partir dos iteráveis <i>it</i> etc.
	<code>s -= z</code>	<code>s.__isub__(z)</code>	<i>s</i> atualizado com a diferença entre <i>s</i> e <i>z</i>
		<code>s.difference_update(it, ...)</code>	<i>s</i> atualizado com a diferença entre <i>s</i> e todos os conjuntos criados a partir dos iteráveis <i>it</i> etc.
		<code>s.symmetric_difference(it)</code>	Complemento de <i>s</i> & set(<i>it</i>)
$s \Delta z$	<code>s ^ z</code>	<code>s.__xor__(z)</code>	Diferença simétrica (complemento da intersecção <i>s</i> & <i>z</i>)
	<code>z ^ s</code>	<code>s.__rxor__(z)</code>	Operador ^ reverso
		<code>s.symmetric_difference_update(it, ...)</code>	<i>s</i> atualizado com a diferença simétrica entre <i>s</i> e todos os conjuntos criados a partir dos iteráveis <i>it</i> etc.
	<code>s ^= z</code>	<code>s.__ixor__(z)</code>	<i>s</i> atualizado com a diferença simétrica de <i>s</i> e <i>z</i>



Quando escrevi este texto, havia um relato de bug de Python – (problema 8743, <http://bugs.python.org/issue8743>) com a seguinte descrição: “Os operadores de `set()` (*or*, *and*, *sub*, *xor* e suas contrapartidas *in-place*) exigem que o parâmetro também seja uma instância de `set()`”, com o efeito colateral indesejado de esses operadores não funcionarem com subclasses de `collections.abc.Set`. O bug já foi corrigido no tronco de Python 2.7 e 3.4 e deverá ser um caso encerrado quando você ler isto.

A tabela 3.3 lista os predicados dos conjuntos: operadores e métodos que devolvem `True` ou `False`.

Tabela 3.3 – Operadores e métodos de comparação de conjuntos que devolvem um booleano

Símbolo matemático	Operador Python	Método	Descrição
		<code>s.isdisjoint(z)</code>	<code>s</code> e <code>z</code> são disjuntos (não têm nenhum elemento em comum)
$e \in S$	<code>e in s</code>	<code>s.__contains__(e)</code>	O elemento <code>e</code> está presente em <code>s</code>
$S \subseteq Z$	<code>s <= z</code>	<code>s.__le__(z)</code>	<code>s</code> é um subconjunto do conjunto <code>z</code>
		<code>s.issubset(it)</code>	<code>s</code> é um subconjunto do conjunto criado a partir do iterável <code>it</code>
$S \subset Z$	<code>s < z</code>	<code>s.__lt__(z)</code>	<code>s</code> é um subconjunto próprio do conjunto <code>z</code>
$S \supseteq Z$	<code>s >= z</code>	<code>s.__ge__(z)</code>	<code>s</code> é um superconjunto do conjunto <code>z</code>
		<code>s.issuperset(it)</code>	<code>s</code> é um superconjunto do conjunto criado a partir do iterável <code>it</code>
$S \supset Z$	<code>s > z</code>	<code>s.__gt__(z)</code>	<code>s</code> é um superconjunto próprio do conjunto <code>z</code>

Além dos operadores e dos métodos derivados da teoria matemática de conjuntos, os conjuntos implementam outros métodos de uso prático sintetizados na tabela 3.4.

Tabela 3.4 – Métodos adicionais de conjuntos

	set	frozenset	
<code>s.add(e)</code>	●		Adiciona o elemento <code>e</code> em <code>s</code>
<code>s.clear()</code>	●		Remove todos os elementos de <code>s</code>
<code>s.copy()</code>	●	●	Cópia rasa de <code>s</code>
<code>s.discard(e)</code>	●		Remove o elemento <code>e</code> de <code>s</code> se estiver presente
<code>s.__iter__()</code>	●	●	Obtém um iterador para <code>s</code>
<code>s.__len__()</code>	●	●	<code>len(s)</code>
<code>s.pop()</code>	●		Remove e devolve um elemento de <code>s</code> , gerando <code>KeyError</code> se <code>s</code> estiver vazio
<code>s.remove(e)</code>	●		Remove o elemento <code>e</code> de <code>s</code> , gerando <code>KeyError</code> se <code>e not in s</code>

Com isso, concluímos nossa visão geral dos recursos relacionados aos conjuntos.

Vamos agora mudar de assunto e discutir como os dicionários e os conjuntos são implementados com tabelas hash. Após ler o restante deste capítulo, você não se surpreenderá mais com o comportamento aparentemente imprevisível exibido por `dict`, `set` e seus parentes.

Por dentro de `dict` e `set`

Entender como os dicionários e conjuntos Python são implementados com tabelas hash ajuda a compreender seus pontos fortes e suas limitações.

Eis algumas perguntas que serão respondidas nesta seção:

- Quão eficientes são os tipos `dict` e `set` em Python?
- Por que eles não são ordenados?
- Por que não podemos usar qualquer objeto Python como chave de `dict` ou como elemento de `set`?
- Por que a ordem das chaves de `dict` ou dos elementos de `set` depende da ordem de inserção e pode mudar durante o tempo de vida da estrutura?
- Por que é ruim incluir itens em um `dict` ou em um `set` durante uma iteração por eles?

Para motivar o estudo das tabelas hash, começaremos exibindo o desempenho incrível de `dict` e de `set` com um teste simples que envolve milhões de itens.

Um experimento para testar o desempenho

Por experiência própria, qualquer pythonista sabe que os dicionários e os conjuntos são rápidos. Confirmaremos esse fato com um experimento controlado.

Para ver como o tamanho de um `dict`, um `set` ou uma `list` afeta o desempenho da busca com o operador `in`, gerei um array com dez milhões de números distintos de ponto flutuante com dupla precisão: o “palheiro” (haystack). Em seguida, gerei um array de “agulhas” (needles): 1.000 números de ponto flutuante sendo que 500 itens foram selecionados do palheiro e 500 com certeza não estão lá.

Para o benchmark de `dict`, usei `dict.fromkeys()` para criar um `dict` chamado `haystack` com 1.000 números de ponto flutuante. Essa foi a preparação para o teste com `dict`. O código propriamente dito, cuja medição de tempo foi feita com o módulo `timeit`, está no exemplo 3.14 (igual ao exemplo 3.11).

Exemplo 3.14 – Busca needles em haystack e conta os itens encontrados

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

O benchmark foi repetido mais quatro vezes; a cada vez, o tamanho de haystack foi aumentado em dez vezes até alcançar o tamanho de 10.000.000 no último teste. O resultado do teste de desempenho de dict está na tabela 3.5.

Tabela 3.5 – Tempo total ao usar o operador in para buscar 1.000 needles em dicts haystack de cinco tamanhos em um laptop Core i7 executando Python 3.4 (os testes mediram o tempo de execução do laço no exemplo 3.14)

tamanho de haystack	Fator	Tempo para dict	Fator
1.000	1x	0,000202s	1,00x
10.000	10x	0,000140s	0,69x
100.000	100x	0,000228s	1,13x
1.000.000	1.000x	0,000290s	1,44x
10.000.000	10.000x	0,000337s	1,67x

Em termos concretos, para verificar a presença de 1.000 chaves de ponto flutuante em um dicionário com 1.000 itens, o tempo de processamento em meu laptop foi de 0,000202s; a mesma busca em um dict com 10.000.000 de itens demorou 0,000337s. Em outras palavras, o tempo por busca no palheiro com 10 milhões de itens foi de 0,337µs para cada agulha – sim, isso é aproximadamente um terço de microssegundo por agulha.

Para comparar, repeti o benchmark com os mesmos palheiros de tamanhos crescentes, porém armazenando haystack como um set ou uma list. Nos testes com set, além de medir o tempo de execução do laço for do exemplo 3.14, medi também o comando de uma linha do exemplo 3.15, que gera o mesmo resultado: conta o número de elementos de needles que também estão em haystack.

Exemplo 3.15 – Uso de intersecção de conjunto para contar as agulhas que estão no palheiro

```
found = len(needles & haystack)
```

A tabela 3.6 mostra os testes lado a lado. Os melhores tempos estão na coluna “Tempo para set&”, que exhibe os resultados para o operador & de conjuntos usando o código do exemplo 3.15. Os piores tempos – conforme esperado – estão na coluna “Tempo para list”, pois não há uma tabela hash para otimizar as buscas com o operador in em uma list, portanto uma varredura completa precisa ser feita, resultando em tempos que aumentam linearmente conforme o tamanho do palheiro.

Tabela 3.6 – Tempo total ao usar o operador in para buscar 1.000 chaves em palheiros de 5 tamanhos, armazenados como dicts, sets e lists em um laptop Core i7 executando Python 3.4 (os testes mediram o tempo de execução do laço do exemplo 3.14, exceto para set&, que usou o exemplo 3.15)

Tamanho de haystack	Fator	Tempo para dict	Fator	Tempo para set	Fator	Tempo para set&	Fator	Tempo para list	Fator
1.000	1x	0,000202s	1,00x	0,000143s	1,00x	0,000087s	1,00x	0,010556s	1,00x
10.000	10x	0,000140s	0,69x	0,000147s	1,03x	0,000092s	1,06x	0,086586s	8,20x
100.000	100x	0,000228s	1,13x	0,000241s	1,69x	0,000163s	1,87x	0,871560s	82,57x
1.000.000	1.000x	0,000290s	1,44x	0,000332s	2,32x	0,000250s	2,87x	9,189616s	870,56x
10.000.000	10.000x	0,000337s	1,67x	0,000387s	2,71x	0,000314s	3,61x	97,948056s	9.278,90x

Se o seu programa faz algum tipo de I/O, o tempo de busca das chaves em dicionários ou em conjuntos será desprezível, independentemente do tamanho do dict ou do set (desde que ele caiba na RAM). Veja o código usado para gerar a tabela 3.6 e a discussão que o acompanha no exemplo A.1 do apêndice A.

Agora que temos evidências concretas da velocidade dos dicionários e dos conjuntos, vamos explorar o que torna isso possível. A discussão sobre o funcionamento interno das tabela hash explica, por exemplo, por que a ordenação de chaves é aparentemente aleatória e instável.

Tabelas hash em dicionários

Apresentaremos uma visão geral de como a linguagem Python utiliza uma tabela hash para implementar um dict. Muitos detalhes foram omitidos – o código de CPython tem alguns truques de otimização⁶ – mas a descrição geral é precisa.



Para simplificar a apresentação a seguir, focaremos primeiro na descrição interna de dict e depois transferiremos os conceitos para os conjuntos.

Uma tabela hash é um array esparsos (ou seja, um array que sempre tem células vazias). Em textos-padrões sobre estrutura de dados, as células em uma tabela hash geralmente são chamadas de “buckets”. Na tabela hash de um dict, há um bucket com dois campos para cada item; esses campos são uma referência para a chave e outra para o valor do item. Como todos os buckets têm o mesmo tamanho, o acesso a um bucket individual é feito pelo offset.

6 O código-fonte do módulo dictobject.c de CPython (<http://hg.python.org/cpython/file/tip/Objects/dictobject.c>) é rico em comentários. Veja também a referência ao livro *Beautiful Code* na seção “Leituras complementares” na página 126.


```

-----
1.0002  01011101011010100001010110111001
        ! !  ! !!! ! !  !! ! !  ! !!!! != 17
1.0003  00001100000111110010000010010110
-----

```

O código que gera o exemplo 3.16 está no apêndice A. A maior parte dele lida com a formatação da saída, porém a listagem completa está no exemplo A.3 para que você possa vê-lo em sua totalidade.



A partir de Python 3.3, um valor aleatório (salt) é somado aos hashes de objetos `str`, `bytes` e `datetime`. O valor de salt é constante em um processo Python, porém varia entre execuções do interpretador. O salt aleatório é uma medida de segurança para evitar um ataque de DOS. Os detalhes estão em uma nota na documentação do método especial `__hash__` (<http://bit.ly/1FESm0m>).

Com essa compreensão básica dos hashes de objetos, estamos prontos para explorar o algoritmo que faz as tabelas hash funcionarem.

Algoritmo de tabela hash

Para acessar o valor em `my_dict[search_key]`, o interpretador Python chama `hash(search_key)` para obter o *valor de hash* de `search_key` e usa os bits menos significativos desse número como offset para procurar um bucket (literalmente, “balde”) na tabela hash (o número de bits usado depende do tamanho da tabela no momento). Se o bucket encontrado estiver vazio, `KeyError` será gerado. Caso contrário, o bucket terá um item – um par `found_key:found_value` – e o interpretador Python verificará se `search_key == found_key`. Se forem iguais, é sinal de que esse era o item procurado; `found_value` será devolvido.

No entanto, se `search_key` e `found_key` não forem iguais, é sinal de que houve uma *colisão de hash*. Isso acontece porque uma função de hash mapeia objetos arbitrários a uma quantidade menor de bits e – além disso – a tabela hash é indexada com um subconjunto desses bits. Para resolver a colisão, o algoritmo pega bits diferentes da hash, mistura-os de determinada maneira e utiliza o resultado como offset para procurar um bucket diferente.⁹ Se ele estiver vazio, `KeyError` será gerado; caso contrário, as chaves serão iguais e o valor do item será devolvido ou então o processo de resolução de colisão será repetido. Veja a figura 3.3, que apresenta um diagrama desse algoritmo.

⁹ A função `C` que embaralha os bits do hash em caso de colisão tem um nome curioso: `perturb`. Para ver todos os detalhes, consulte `dictobject.c` no código-fonte de CPython (<http://bit.ly/1JzB8rA>).

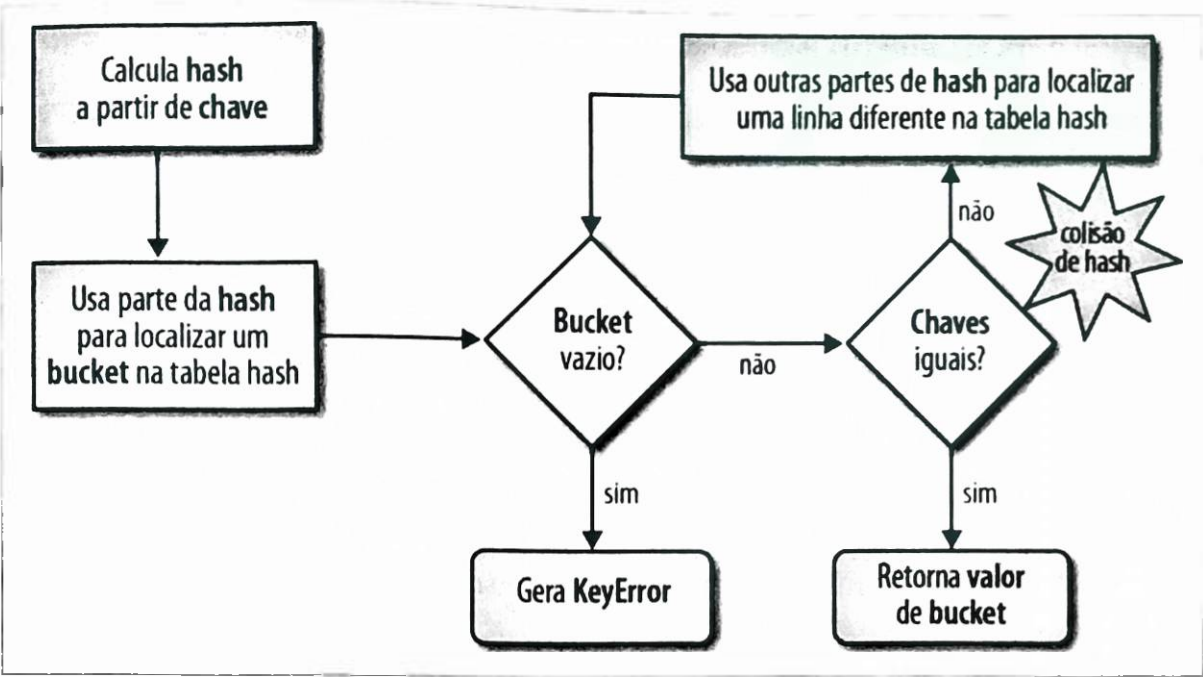


Figura 3.3 – Fluxograma para obter um item de um dict; dada uma chave, esse procedimento devolve um valor ou gera KeyError.

O processo para inserir ou atualizar um item é o mesmo, exceto que, quando um bucket vazio for localizado, o novo item será colocado aí e, quando um bucket com uma chave correspondente for encontrado, o valor desse bucket será sobrescrito com o novo valor.

Além disso, ao inserir itens, o interpretador Python pode decidir que a tabela hash está muito cheia e poderá recriá-la em um novo local, com mais espaço. À medida que a tabela hash cresce, o mesmo ocorre com a quantidade de bits de hash usados para os offsets de buckets, e isso mantém a taxa de colisão baixa.

Essa implementação pode parecer trabalhosa, mas, mesmo com milhões de itens em um dict, muitas buscas ocorrem sem que haja colisão e o número médio de colisões por busca permanece entre um e dois. No uso normal, até mesmo as chaves mais arduas podem ser encontradas após algumas poucas colisões terem sido resolvidas.

Conhecer a implementação interna de dict pode explicar os pontos fortes e as limitações dessa estrutura de dados e de todas as demais estruturas que derivam dela em Python. Agora estamos prontos para considerar por que os dicts Python comportam-se da maneira como o fazem.

Consequências práticas de como os dicionários funcionam

Nas subseções a seguir, discutiremos as limitações e as vantagens que a implementação de tabela hash proporciona ao uso de dict.

As chaves precisam ser objetos hashable.

Um objeto é hashable se todos os requisitos a seguir forem atendidos:

1. Oferece suporte à função `hash()` por meio de um método `__hash__()` que sempre devolve o mesmo valor durante o tempo de vida do objeto.
2. Oferece suporte à igualdade por meio do método `__eq__()`.
3. Se `a == b` é `True`, então `hash(a) == hash(b)` também deve ser `True`.

Os tipos definidos pelo usuário são hashable por padrão, pois seus valores de hash são os seus `id()` e, na comparação, todos são diferentes.



Se você implementar uma classe com um método `__eq__` personalizado, deverá implementar também um `__hash__` adequado, pois sempre deverá garantir que se `a == b` é `True`, então `hash(a) == hash(b)` também será `True`. Caso contrário, você estará infringindo uma invariante do algoritmo de tabelas hash, com uma consequência grave: os dicionários e os conjuntos não lidarão com seus objetos de forma confiável. Se um `__eq__` personalizado depender de um estado mutável, `__hash__` deverá gerar `TypeError` com uma mensagem como `unhashable type: 'MyClass'`.

dicts têm overhead significativo de memória

Pelo fato de um `dict` usar uma tabela hash internamente e as tabelas hash terem de ser esparsas para funcionar, eles não são eficientes quanto ao espaço. Por exemplo, se você estiver lidando com uma grande quantidade de registros, fará sentido armazená-los em uma lista de tuplas ou de `namedtuple` em vez de usar uma lista de dicionários em estilo JSON com um `dict` por registro. Substituir dicionários por tuplas reduz o uso de memória de duas formas: elimina o overhead de uma tabela hash por registro e evita armazenar os nomes dos campos novamente a cada registro.

Para os tipos definidos pelo usuário, o atributo de classe `__slots__` altera a armazenagem de atributos de instâncias, de um `dict` para uma tupla a cada instância. Isso será discutido na seção “Economizando espaço com o atributo de classe `__slots__`” na página 307 (Capítulo 9).

Tenha em mente que estamos falando de otimizações de espaço. Se estiver lidando com alguns milhões de objetos e seu computador tiver gigabytes de RAM, você deverá adiar esse tipo de otimização até que sejam realmente necessários. A otimização é o altar em que a facilidade de manutenção é sacrificada.

Buscar uma chave é muito rápido

A implementação de `dict` é uma troca de espaço por tempo: os dicionários têm *overhead* significativo de memória, porém proporcionam um acesso rápido, independentemente do tamanho do dicionário – desde que ele caiba na memória. Como mostra a tabela 3.5, quando aumentamos o tamanho de um `dict` de 1.000 para 10.000.000 de elementos, o tempo de busca aumentou de um fator de 2,8, passando de 0,000163s para 0,000456s. Esse último valor revela que poderíamos buscar mais de dois milhões de chaves por segundo em um `dict` com dez milhões de itens.

A ordem das chaves depende da ordem de inserção

Quando uma colisão de hash ocorre, a segunda chave acaba em uma posição que não ocuparia normalmente se tivesse sido inserida antes. Sendo assim, um `dict` criado como `dict([(key1, value1), (key2, value2)])` será comparado como igual a `dict([(key2, value2), (key1, value1)])`, porém a ordem de suas chaves poderá não ser a mesma se os hashes de `key1` e de `key2` colidirem.

O exemplo 3.17 mostra o efeito de carregar três dicionários com os mesmos dados, mas em ordem diferente. Os dicionários resultantes são todos comparados como iguais, apesar de sua ordem não ser a mesma.

Exemplo 3.17 – `dialcodes.py` preenche três dicionários com os mesmos dados ordenados de maneiras diferentes

```
# códigos de discagem dos dez países mais populosos
```

```
DIAL_CODES = [  
    (86, 'China'),  
    (91, 'India'),  
    (1, 'United States'),  
    (62, 'Indonesia'),  
    (55, 'Brazil'),  
    (92, 'Pakistan'),  
    (880, 'Bangladesh'),  
    (234, 'Nigeria'),  
    (7, 'Russia'),  
    (81, 'Japan'),  
]
```

```
d1 = dict(DIAL_CODES) ❶
```

```
print('d1:', d1.keys())
```

```
d2 = dict(sorted(DIAL_CODES)) ❷
```

```
print('d2:', d2.keys())
```

```
d3 = dict(sorted(DIAL_CODES, key=lambda x:x[1])) ❸
print('d3:', d3.keys())
assert d1 == d2 and d2 == d3 ❹
```

- ❶ d1: criado a partir de tuplas em ordem decrescente de população do país.
- ❷ d2: preenchido com tuplas ordenadas de acordo com o código de discagem.
- ❸ d3: carregado com tuplas ordenadas de acordo com o nome do país.
- ❹ Os dicionários são comparados como iguais, pois armazenam os mesmos pares `key:value`.

O exemplo 3.18 mostra a saída.

Exemplo 3.18 – A saída de `dialcodes.py` mostra três ordens diferentes de chaves

```
d1: dict_keys([880, 1, 86, 55, 7, 234, 91, 92, 62, 81])
d2: dict_keys([880, 1, 91, 86, 81, 55, 234, 7, 92, 62])
d3: dict_keys([880, 81, 1, 86, 55, 7, 234, 91, 92, 62])
```

Adicionar itens em um dicionário pode alterar a ordem das chaves existentes

Sempre que um novo item é adicionado em um `dict`, o interpretador Python pode decidir que a tabela hash desse dicionário precisa crescer. Isso implica criar uma nova tabela hash maior e adicionar todos os itens atuais à nova tabela. Durante esse processo, poderão ocorrer novas (porém diferentes) colisões de hash e, como resultado, é bem provável que as chaves sejam ordenadas de modo diferente na nova tabela hash. Tudo isso depende da implementação, portanto não é possível prever exatamente quando isso ocorrerá. Se você estiver fazendo uma iteração pelas chaves do dicionário e estiver alterando-as ao mesmo tempo, seu laço poderá não percorrer todos os itens conforme esperado – nem mesmo aqueles que já estavam no dicionário antes dos acréscimos.

É por isso que modificar o conteúdo de um `dict` durante uma iteração não é uma boa ideia. Se for necessário percorrer e acrescentar itens a um dicionário, faça isso em dois passos: leia o `dict` do início ao fim e reúna os acréscimos necessários em um segundo `dict`. Em seguida, atualize o primeiro `dict` com o segundo usando o método `.update()`.



Em Python 3, os métodos `.keys()`, `.items()` e `.values()` devolvem *views* de dicionários, que se comportam mais como conjuntos do que como as listas devolvidas por esses métodos em Python 2. Essas *views* também são dinâmicas: elas não duplicam o conteúdo do `dict` e refletem imediatamente qualquer mudança feita em `dict`.

Agora podemos aplicar o que sabemos sobre as tabelas hash aos conjuntos.

Como os conjuntos funcionam – consequências práticas

Os tipos `set` e `frozenset` também são implementados com uma tabela hash, exceto que cada bucket armazena somente uma referência ao elemento (como se fosse uma chave em um `dict`, mas sem um valor para acompanhá-la). De fato, antes de `set` ter sido acrescentado à linguagem, geralmente usávamos dicionários com valores descartáveis somente para testar rapidamente a presença de chaves.

Tudo que foi dito na seção “Consequências práticas de como os dicionários funcionam” na página 121 sobre como a tabela hash subjacente determina o comportamento de um `dict` aplica-se a um `set`. Sem repetir a seção anterior, podemos resumir-la para os conjuntos em apenas algumas palavras:

- Os elementos dos conjuntos devem ser objetos hashable.
- Os conjuntos têm um overhead significativo de memória.
- O teste de presença é muito eficiente.
- A ordem dos elementos depende da ordem de inserção.
- Adicionar elementos a um conjunto pode alterar a ordem de outros elementos.

Resumo do capítulo

Dicionários são uma pedra angular de Python. Além do `dict` básico, a biblioteca-padrão oferece mapeamentos especializados práticos e prontos para uso como `defaultdict`, `OrderedDict`, `ChainMap` e `Counter`, todos definidos no módulo `collections`. O mesmo módulo também oferece a classe `UserDict`, fácil de ser estendida.

Dois métodos poderosos disponíveis na maioria dos mapeamentos são `setdefault` e `update`. O método `setdefault` é usado para atualizar itens que armazenem valores mutáveis, por exemplo, um `dict` de valores `list`, para evitar buscas redundantes à mesma chave. O método `update` permite inserção em bloco ou a sobrescrita de itens a partir de qualquer outro mapeamento, de iteráveis que forneçam pares (`key`, `value`) ou de argumentos nomeados. Os construtores dos mapeamentos também usam `update` internamente, permitindo que as instâncias sejam inicializadas a partir de mapeamentos, iteráveis ou argumentos nomeados.

Um ponto de extensão inteligente na API de mapeamento é o método `__missing__`, que permite personalizar o que acontece quando uma chave não é encontrada.

O módulo `collections.abc` oferece as classes-base abstratas `Mapping` e `MutableMapping` como referências e para verificação de tipos. O pouco conhecido `MappingProxyType` do módulo `types` cria mapeamentos imutáveis. Há também ABCs para `Set` e `MutableSet`.

A implementação de tabela hash subjacente a `dict` e `set` é extremamente rápida. Entender sua lógica explica por que os itens são aparentemente desordenados e podem até mesmo mudar de ordem quando não estamos olhando. Há um preço a pagar por toda essa velocidade, e esse preço é a memória.

Leituras complementares

Em *The Python Standard Library* (Biblioteca-Padrão do Python), a seção 8.3. *collections – Container datatypes* (<https://docs.python.org/3/library/collections.html>) inclui exemplos e receitas práticas com diversos tipos de mapeamento. O código-fonte Python do módulo `Lib/collections/__init__.py` é uma ótima referência para qualquer pessoa que queira criar um novo tipo de mapeamento ou apreender a lógica dos tipos existentes.

O capítulo 1 do livro *Python Cookbook*, 3ª edição (O'Reilly)¹⁰ de David Beazley e Brian K. Jones tem vinte receitas práticas e esclarecedoras com estruturas de dados – a maior parte delas usando `dict` de formas espertas.

O capítulo 18, “Python's Dictionary Implementation: Being All Things to All People”, do livro *Beautiful Code* (O'Reilly, <http://oreil.ly/1LGCRlr>) – escrito por A. M. Kuchling, mantenedor de Python e autor de muitas páginas da documentação e de vários how-tos oficiais – inclui uma explicação detalhada do funcionamento interno de `dict` em Python. Além disso, há muitos comentários no código-fonte do módulo `dictobject.c` de CPython (<http://hg.python.org/cpython/file/tip/Objects/dictobject.c>). A apresentação *The Mighty Dictionary* (<http://bit.ly/1JzEjiR>) de Brandon Craig Rhodes é excelente e mostra como as tabelas hash funcionam usando vários slides com... tabelas!

A justificativa para adicionar conjuntos à linguagem está documentada na PEP 218 – *Adding a Built-In Set Object Type* (Acrescentando um tipo de objeto embutido `set`, <https://www.python.org/dev/peps/pep-0218/>). Quando a PEP 218 foi aprovada, nenhuma sintaxe literal especial foi adotada para conjuntos. Os literais `set` foram criados para Python 3 e portados para a versão Python 2.7, juntamente com as `dict` e as `set` comprehensions. A PEP 274 – *Dict Comprehensions* (<https://www.python.org/dev/peps/pep-0274/>) é a certidão de nascimento das `dictcomps`. Não consegui encontrar uma PEP para as `setcomps`; aparentemente, elas foram adotadas porque se parecem muito com suas irmãs – o que é um excelente motivo.

¹⁰ N.T.: Tradução brasileira publicada pela Novatec Editora (<http://novatec.com.br/livros/python-cookbook/>).

Ponto de vista

Meu amigo Geraldo Cohen certa vez comentou que a linguagem Python é “simples e correta”.

O tipo `dict` é um exemplo de simplicidade e corretude. É altamente otimizado para realizar bem uma única tarefa: recuperar chaves arbitrárias. É rápido e robusto o suficiente para ser usado na construção do próprio interpretador Python. Se você precisa de uma ordenação previsível, utilize `OrderedDict`. Esse não é um requisito na maioria dos usos de mapeamentos, portanto faz sentido manter a implementação básica simples e oferecer variações na biblioteca-padrão.

Compare com a linguagem PHP, em que os arrays são descritos da seguinte maneira no Manual de PHP (http://php.net/manual/pt_BR/language.types.array.php):

Um array no PHP é atualmente¹¹ um mapa ordenado. Um mapa é um tipo que relaciona *valores* para *chaves*. Este tipo é otimizado de várias maneiras, então você pode usá-lo como um array real, ou uma lista (vetor), hashtable (que é uma implementação de mapa), dicionário, coleção, pilha, fila e provavelmente mais.

A partir dessa descrição, não sei qual é o verdadeiro custo de usar o híbrido `list/OrderedDict` em PHP.

O objetivo deste capítulo e do anterior foi mostrar os tipos de coleção de Python otimizados para usos específicos. Mostrei que, além dos confiáveis `list` e `dict`, há alternativas especializadas para casos de uso diferentes.

Antes de conhecer Python, eu havia feito programação web com Perl, PHP e JavaScript. Eu realmente gostava de ter uma sintaxe literal para mapeamentos nessas linguagens, e sinto muita falta disso sempre que preciso usar Java ou C. Uma boa sintaxe literal para mapeamentos facilita criar configurações, fazer implementações baseadas em tabelas (table-driven) e armazenar dados para prototipação e testes. A falta disso pressionou a comunidade Java a adotar XML como formato de dados, mas o padrão XML é extenso e excessivamente complexo.

O formato JSON foi proposto como “*The Fat-Free Alternative to XML*” (A alternativa sem gorduras ao XML, <http://www.json.org/fatfree.html>) e tornou-se um enorme sucesso, substituindo o XML em diversos contextos. Uma sintaxe concisa para listas e dicionários é um excelente formato para intercâmbio de dados.

11 Nota do autor/revisor: A versão pt_BR do Manual do PHP traduz “actually” como “atualmente”, o que é um erro. Até o Google Translate faz melhor: ele sugere “na realidade”. Por outro lado, pelo menos existe uma tradução da documentação oficial do PHP. Vergonhosamente, não temos uma tradução da documentação de Python, nem mesmo uma tradução ruim.

PHP e Ruby imitaram a sintaxe de hash usada em Perl adotando => para associar chaves a valores.

JavaScript seguiu a linha de Python e utiliza :. É claro que o formato JSON surgiu de JavaScript, mas, por acaso, ele também é quase um subconjunto exato da sintaxe de Python. JSON é compatível com Python, exceto pela grafia dos valores true, false e null. A sintaxe que hoje todos usam para intercâmbio de dados é a sintaxe de dict e de list em Python.

Simples e correta.

CAPÍTULO 4

Texto versus bytes

Seres humanos usam texto. Computadores falam bytes.¹

— Esther Nam e Travis Fischer

Character Encoding and Unicode in Python

Python 3 introduziu uma distinção clara entre strings de texto humano e sequências de bytes em estado bruto. A conversão implícita de sequências de bytes para texto Unicode é coisa do passado. Este capítulo lida com strings Unicode, sequências binárias e as codificações usadas para a conversão entre elas.

Conforme o contexto de seu trabalho com Python, um entendimento mais profundo de Unicode poderá ou não ser de importância vital para você. No final, a maior parte dos problemas discutidos neste capítulo não afeta programadores que lidem somente com texto ASCII. Contudo, mesmo que esse seja o seu caso, não há escapatória da divisão *str* versus *byte*. Como bônus, você perceberá que os tipos especializados de sequências binárias oferecem recursos que o tipo *str* “usado para todos os propósitos” de Python 2 não tem.

Neste capítulo, discutiremos os seguintes assuntos:

- caracteres, códigos Unicode e representações em bytes;
- recursos exclusivos de sequências binárias: `bytes`, `bytearray` e `memoryview`;
- codecs para o Unicode completo e conjuntos legados de caracteres;
- como evitar e tratar erros de codificação;
- melhores práticas ao lidar com arquivos-texto;
- a armadilha da codificação default e problemas de E/S padrão (standard I/O);

¹ Slide 12 da apresentação “Character Encoding and Unicode in Python” (Codificação de caracteres e Unicode em Python) na PyCon 2014 (slides em <http://bit.ly/1JzF1MY>, vídeo em <http://bit.ly/1JzF37P>).

- comparações seguras de texto Unicode com normalização;
- funções utilitárias para normalização, case folding e remoção de diacríticos à força;
- ordenação correta de texto Unicode com locale e a biblioteca PyUCA;
- metadados de caracteres na base de dados Unicode;
- APIs de modo dual que lidam com `str` e `bytes`.

Vamos começar pelos caracteres, códigos Unicode e bytes.

Falhas de caracteres

O conceito de “string” é bem simples: uma string é uma sequência de caracteres. O problema está na definição de “caractere”.

Atualmente, a melhor definição de “caractere” que temos é um caractere Unicode. Assim, os itens em uma `str` em Python 3 são caracteres Unicode, assim como os itens de um objeto `unicode` em Python 2 – e não os bytes puros obtidos de uma `str` em Python 2.

O padrão Unicode separa explicitamente a identidade de um caractere de suas possíveis representações em bytes:

- A identidade de um caractere – o *código Unicode* (code point) – é um número de 0 a 1.114.111 (base 10), exibido nos documentos do padrão Unicode na forma de um prefixo “U+” seguido de 4 a 6 dígitos hexadecimais. Por exemplo, o código Unicode para a letra A é U+0041, o símbolo de euro é U+20AC e o símbolo musical da clave de sol é atribuído ao code point U+1D1E. Aproximadamente 10% dos code points válidos têm caracteres atribuídos a eles em Unicode 6.3, que é o padrão usado em Python 3.4.
- Os bytes propriamente ditos que representam um caractere dependem da *codificação* (encoding) usada. Uma codificação é um algoritmo que converte códigos Unicode em sequências de bytes e vice-versa. O código Unicode para A (U+0041) está codificado como o byte único `\x41` na codificação UTF-8 ou como os bytes `\x41\x00` na codificação UTF-16LE. Como outro exemplo, o símbolo de euro (U+20AC) tem três bytes em UTF-8 – `\xe2\x82\xac` –, mas em UTF-16LE, é codificado com dois bytes: `\xac\x20`.

Converter de códigos Unicode para bytes é *codificar* (encode); converter de bytes para códigos Unicode é *decodificar* (decode). Veja o exemplo 4.1.

Exemplo 4.1 – Codificação e decodificação

```
>>> s = 'café'
>>> len(s) # ❶
4
>>> b = s.encode('utf8') # ❷
>>> b
b'caf\xc3\xa9' # ❸
>>> len(b) # ❹
5
>>> b.decode('utf8') # ❺
'café'
```

- ❶ A str 'café' tem quatro caracteres Unicode.
- ❷ Codificamos str para bytes usando a codificação UTF-8.
- ❸ Literais do tipo bytes começam com um prefixo b.
- ❹ O objeto bytes na variável b tem cinco bytes (o codepoint do caractere “é” é codificado com dois bytes em UTF-8).
- ❺ Decodificamos bytes para str usando a codificação UTF-8.



Se precisar de ajuda para se lembrar da distinção entre `.decode()` e `.encode()`, convença a si mesmo de que as sequências de bytes podem ser core dumps enigmáticos do computador, enquanto objetos str Unicode são textos para “seres humanos”. Desse modo, faz sentido que *decodifiquemos* bytes para str usando `.decode()` para obter um texto legível aos seres humanos, e que *codifiquemos* de str para bytes com `.encode()` para armazenamento ou transmissão.

Embora o tipo str em Python 3 seja praticamente o tipo unicode de Python 2 com um novo nome, o tipo bytes de Python 3 não é apenas o velho tipo str renomeado; além disso, há também o tipo bytearray, parente próximo do tipo bytes. Portanto vale a pena dar uma olhada nas sequências binárias antes de passar para os problemas de codificação/decodificação.

0 essencial sobre bytes

Os novos tipos para sequências binárias são diferentes do str de Python 2 em diversos aspectos. O primeiro ponto a saber é que há dois tipos embutidos básicos para sequências binárias: o tipo bytes imutável introduzido em Python 3 e o bytearray mutável, acrescentado em Python 2.6. (Python 2.6 também introduziu bytes, porém é somente um apelido [alias] para o tipo str e não se comporta como o tipo bytes em Python 3.)

Cada item em bytes ou em bytearray é um inteiro de 0 a 255, e não uma string de um caractere como temos nas str em Python 2. No entanto uma fatia de uma sequência binária sempre gera uma sequência binária de mesmo tipo – incluindo fatias de tamanho 1. Veja o exemplo 4.2.

Exemplo 4.2 – Uma sequência de cinco bytes como bytes e como bytearray

```
>>> cafe = bytes('café', encoding='utf_8') ❶
>>> cafe
b'caf\xc3\xa9'
>>> cafe[0] ❷
99
>>> cafe[:1] ❸
b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr ❹
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:] ❺
bytearray(b'\xa9')
```

- ❶ bytes pode ser construído a partir de uma str, dada uma codificação.
- ❷ Cada item é um inteiro em range(256).
- ❸ Fatias de bytes também são bytes – mesmo as fatias com um único byte.
- ❹ Não existe uma sintaxe literal para bytearray: eles são mostrados como bytearray(), com um literal bytes como argumento.
- ❺ Uma fatia de bytearray também é um bytearray.



O fato de `my_bytes[0]` ser um `int`, mas `my_bytes[:1]` devolver um objeto bytes de tamanho 1 não deve ser nenhuma surpresa. O único tipo de sequência em que `s[0] == s[:1]` é o tipo `str`. Embora seja prático, esse comportamento de `str` é excepcional. Para todas as demais sequências, `s[i]` devolve um item e `s[i:i+1]` devolve uma sequência de mesmo tipo com o item `s[i]` dentro dela.

Apesar de as sequências binárias serem realmente sequências de inteiros, sua notação literal foi escolhida para tornar legível algum texto ASCII embutido nelas. Assim, três formas diferentes de exibição são usadas de acordo com o valor de cada byte:

- Para bytes no intervalo ASCII possível de ser exibido – do espaço ao ~ – o próprio caractere ASCII é usado.
- Para bytes correspondentes a tabulação, quebra de linha, carriage return e \, as sequências de escape `\t`, `\n`, `\r` e `\\` são usadas.

- Para todos os demais valores de byte, uma sequência de escape hexadecimal é usada (por exemplo, `\x00` é o byte nulo).

É por isso que, no exemplo 4.2, vemos `b'caf\xc3\xa9'`: os três primeiros bytes `b'caf'` estão no intervalo ASCII que podem ser mostrados, e os dois últimos não estão.

Tanto `bytes` quanto `bytearray` aceitam todos os métodos de `str`, exceto aqueles que fazem formatação (`format`, `format_map`) e outros que dependem de dados Unicode, incluindo `casefold`, `isdecimal`, `isidentifier`, `isnumeric`, `isprintable` e `encode`. Isso quer dizer que você pode usar métodos conhecidos de string como `endwith`, `replace`, `strip`, `translate`, `upper` e dezenas de outros com sequências binárias – apenas usando argumentos do tipo `bytes` no lugar de `str`. Além disso, as funções de expressão regular do módulo `re` também funcionam com sequências binárias se a `regex` for compilada a partir de uma sequência binária em vez de uma `str`. O operador `%` não funciona com sequências binárias em Python de 3.0 a 3.4, mas deverá ser aceito na versão 3.5 de acordo com a *PEP 461 – Adding % formatting to bytes and bytearray* (Acrescentando formatação com `%` a `bytes` e `bytearrays`, <https://www.python.org/dev/peps/pep-0461/>).

As sequências binárias têm um método de classe chamado `fromhex` que `str` não tem; esse método cria uma sequência binária interpretando pares de dígitos hexadecimais opcionalmente separados com espaços:

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

Outra maneira de criar instâncias de `bytes` ou de `bytearray` é chamar seus construtores com:

- Uma `str` e um argumento nomeado `encoding`.
- Um iterável que forneça itens com valores de 0 a 255.
- Um único inteiro para criar uma sequência binária com esse tamanho, inicializada com bytes nulos. (Essa forma se tornará obsoleta em Python 3.5 e será removida em Python 3.6). Consulte a *PEP 467 — Minor API improvements for binary sequences* (Pequenas melhorias na API para sequências binárias, <https://www.python.org/dev/peps/pep-0467/>).
- Um objeto que implemente o protocolo de buffer (por exemplo, `bytes`, `bytearray`, `memoryview`, `array.array`); os bytes são copiados do objeto de origem para a sequência binária recém-criada.

Criar uma sequência binária a partir de um objeto buffer ou similar é uma operação de baixo nível que poderá envolver casting de tipo. Veja uma demonstração no exemplo 4.3.

Exemplo 4.3 – Inicializando bytes a partir dos dados brutos de um array

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
>>> octets = bytes(numbers) ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

- ❶ O typecode 'h' cria um array de short integers (16 bits).
- ❷ octets armazena uma cópia dos bytes que compõem numbers.
- ❸ Esses são os 10 bytes que representam os cinco short integers.

Criar um objeto bytes ou bytearray a partir de qualquer origem que seja um buffer ou algo similar sempre fará os bytes serem copiados. Em comparação, os objetos `memoryview` permitem compartilhar memória entre estruturas de dados binárias. Para extrair informações estruturadas de sequências binárias, o módulo `struct` é muito prático. Veremos esse módulo trabalhar em conjunto com bytes e `memoryview` na próxima seção.

Structs e memory views

O módulo `struct` oferece funções para análise de bytes compactos, extraíndo-os em uma tupla com campos de diferentes tipos, e para realizar a conversão inversa, de uma tupla para bytes compactos. `struct` é usado com objetos bytes, bytearray e `memoryview`.

Como vimos na seção “Memory Views” na página 78, a classe `memoryview` não permite criar nem armazenar sequências de bytes, porém oferece um acesso de memória compartilhada a fatias de dados de outras sequências binárias, arrays compactos e buffers como as imagens PIL (Python Imaging Library)², sem copiar os bytes.

O exemplo 4.4 mostra o uso de `memoryview` e de `struct` juntos para extrair a largura e a altura de uma imagem GIF.

Exemplo 4.4 – Usando `memoryview` e `struct` para inspecionar o cabeçalho de uma imagem GIF

```
>>> import struct
>>> fmt = '<3s3sHH' # ❶
>>> with open('filter.gif', 'rb') as fp:
...     img = memoryview(fp.read()) # ❷
...
>>> header = img[:10] # ❸
>>> bytes(header) # ❹
b'GIF89a+\x02\xe6\x00'
```

² Pillow (<https://pillow.readthedocs.org/en/latest/>) é a versão ramificada (fork) mais ativa de PIL.

```
>>> struct.unpack(fmt, header) # ❸
(b'GIF', b'89a', 555, 230)
>>> del header # ❹
>>> del img
```

- ❶ formato de struct: < *little-endian*; 3s3s duas sequências de 3 bytes; HH dois inteiros de 16 bits.
- ❷ Cria a `memoryview` a partir do conteúdo do arquivo em memória...
- ❸ ... e depois outra `memoryview` ao fatiar a primeira; nenhum byte é copiado nesse caso.
- ❹ Converte para bytes somente para exibição; 10 bytes são copiados aqui.
- ❺ Desempacota `memoryview` em uma tupla com: tipo, versão, largura e altura.
- ❻ Apaga as referências para liberar a memória associada às instâncias de `memoryview`.

Observe que o fatiamento de uma `memoryview` devolve uma nova `memoryview`, sem copiar bytes [Leonardo Rochael – um dos revisores técnicos – destacou que copiaríamos menos bytes se usássemos o módulo `mmap` para abrir a imagem como um arquivo mapeado em memória. Não discutirei o módulo `mmap` neste livro, mas, se você lê e altera arquivos binários com frequência, será muito produtivo estudar a documentação de *mmap – Memory-mapped file support* (`mmap` – Suporte a arquivos mapeados em memória, <https://docs.python.org/3/library/mmap.html>)].

Não vamos nos aprofundar na `memoryview` nem no módulo `struct` neste livro, porém, se você trabalha com dados binários, vale a pena estudar a documentação: *Built-in Types » Memory Views* (Tipos embutidos » Memory Views, <http://bit.ly/1Vm7ZnI>) e *struct — Interpret bytes as packed binary data* (`struct` – Interpretar bytes como dados binários compactos, <http://bit.ly/1Vm7YjA>).

Após essa rápida exploração das sequências binárias em Python, vamos ver como elas são convertidas de/para strings.

Codificadores/decodificadores básicos

A distribuição Python inclui mais de cem codecs (codificador/decodificador) para conversão de texto em byte e vice-versa. Todo codec tem um nome, por exemplo, `'utf_8'`, e, muitas vezes, tem apelidos como `'utf8'`, `'utf-8'` e `'U8'`, que você pode usar como o argumento `encoding` em funções como `open()`, `str.encode()`, `bytes.decode()` e assim por diante. O exemplo 4.5 mostra o mesmo texto codificado como três sequências de bytes diferentes.

Exemplo 4.5 – A string “El Niño” codificada com três codecs gerando seqüências bem diferentes de bytes

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8 b'El Ni\xc3\xb1o'
utf_16 b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xfi\x00o\x00'
```

A figura 4.1 mostra uma variedade de codecs gerando bytes a partir de caracteres como a letra “A” até o símbolo musical da clave de sol. Observe que as três últimas codificações são codificações multibyte de tamanho variável.

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
À	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
€	U+06BF	*	*	*	*	*	DA BF	BF 06
ˆ	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
ƒ	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+8C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
𐀀	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
♯	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

Figura 4.1 – Doze caracteres, seus códigos Unicode e a representação em bytes (em hexadecimal) em sete codificações diferentes (asteriscos indicam que o caractere não pode ser representado nessa codificação).

Todos aqueles asteriscos da figura 4.1 deixam evidente que algumas codificações, como ASCII e até mesmo o GB2312 (que usa múltiplos bytes), não são capazes de representar todos os caracteres Unicode. As codificações UTF, porém, foram criadas para tratar todos os códigos Unicode.

As codificações mostradas na figura 4.1 foram escolhidas como uma amostra representativa:

latin1 também conhecida como iso8859_1

É importante porque é a base de outras codificações, como cp1252 e o próprio Unicode (observe como os valores dos bytes em latin1 aparecem nos bytes em cp1252 e até nos códigos Unicode).

cp1252

É um superconjunto de `latin1` criado pela Microsoft, acrescentando símbolos úteis como aspas curvas e o símbolo de € (euro); alguns aplicativos do Windows chamam essa codificação de “ANSI”, mas ela nunca foi um padrão ANSI de verdade.

cp437

É o conjunto original de caracteres do IBM PC, com caracteres para desenhar caixas. Incompatível com `latin1`, que surgiu depois.

gb2312

Padrão legado para codificar os ideogramas chineses simplificados usados na China continental; uma das várias codificações multibyte amplamente utilizadas em línguas asiáticas.

utf-8

De longe, a codificação de 8 bits mais comum na Web³; retrocompatível com o ASCII (textos ASCII puro são textos válidos em UTF-8).

utf-16le

Uma forma do esquema de codificação UTF-16 de 16 bits; todas as codificações UTF-16 aceitam códigos Unicode além de U+FFFF por meio de sequências de escape chamadas “pares substitutos” (surrogate pairs).



UTF-16 substituiu a codificação original de 16 bits do Unicode 1.0 – UCS-2 – muito tempo atrás, em 1996. UCS-2 continua em uso em diversos sistemas, porém aceita códigos Unicode somente até U+FFFF. Em Unicode 6.3, mais de 50% dos códigos Unicode alocados estão acima de U+FFFF, incluindo os cada vez mais populares pictogramas para emoji.

Com essa apresentação geral das codificações comuns agora concluída, vamos passar para o tratamento de problemas em operações de codificação e de decodificação.

Entendendo os problemas de codificação/decodificação

Embora exista uma exceção `UnicodeError` genérica, o erro informado quase sempre é mais específico: será um `UnicodeEncodeError` (na conversão de `str` para sequências binárias) ou um `UnicodeDecodeError` (ao ler sequências binárias para uma `str`). Carregar

³ Em setembro de 2014, a *W3Techs: Usage of Character Encodings for Websites* (Uso de codificações de caracteres em sites, <http://bit.ly/w3techs-en>) afirmou que 81,4% dos sites usam UTF-8, enquanto a *Built With: Encoding Usage Statistics* (Estatísticas do uso de codificações, <http://trends.builtwith.com/encoding>) estima esse valor em 79,4%.

módulos Python também pode gerar um `SyntaxError` quando a codificação código-fonte for inesperada. Mostraremos como lidar com todos esses erros nas próximas seções.



O primeiro ponto a ser observado quando você obtiver um erro de Unicode é o tipo exato da exceção. É um `UnicodeEncodeError`, um `UnicodeDecodeError` ou outro erro (por exemplo, `SyntaxError`) que menciona um problema de codificação? Para solucionar o problema, você deverá entendê-lo antes.

Lidando com `UnicodeEncodeError`

A maioria dos codecs que não é UTF trata somente um pequeno subconjunto dos caracteres Unicode. Na conversão de texto para bytes, se um caractere não estiver definido na codificação-alvo, `UnicodeEncodeError` será gerado, a menos que um tratamento especial seja oferecido passando um argumento `errors` ao método ou à função de codificação. O exemplo 4.6 mostra como funcionam as diferentes formas de tratamento de erro.

Exemplo 4.6 – Codificação para bytes: sucesso e tratamento de erro

```
>>> city = 'São Paulo'
>>> city.encode('utf_8') ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1') ❷
b'S\xe3o Paulo'
>>> city.encode('cp437') ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../lib/python3.4/encodings/cp437.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore') ❹
b'So Paulo'
>>> city.encode('cp437', errors='replace') ❺
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace') ❻
b'S&#227;o Paulo'
```

❶ As codificações 'utf_?' tratam qualquer str.

- ❷ 'iso8859_1' também funciona para a str 'São Paulo'.
- ❸ 'cp437' não é capaz de codificar 'ã' ("a" com til). O tratamento de erro default – 'strict' – gera `UnicodeEncodeError`.
- ❹ O tratamento `error='ignore'` descarta silenciosamente os caracteres que não podem ser codificados; normalmente, não é uma boa ideia.
- ❺ Na codificação, `error='replace'` substitui os caracteres que não podem ser codificados por '?'; dados são perdidos, mas os usuários saberão que há algo faltando.
- ❻ 'xmlcharrefreplace' substitui caracteres que não podem ser codificados por uma entidade XML.



O tratamento de erro de codecs é extensível. Você pode registrar strings extras para o argumento `errors` passando um nome e uma função de tratamento de erro à função `codecs.register_error`. Veja a documentação de `codecs.register_error` (<http://bit.ly.1Vm83DZ>).

Lidando com `UnicodeDecodeError`

Nem todo byte armazena um caractere ASCII válido, e nem toda sequência de bytes é um UTF-8 ou um UTF-16 válido; desse modo, quando pressupomos uma dessas codificações ao converter uma sequência binária para texto, teremos um `UnicodeDecodeError` se algum byte inesperado for encontrado.

Por outro lado, muitas codificações legadas de 8 bits como 'cp1252', 'iso8859_1' e 'koi8_r' podem decodificar qualquer sequência de bytes, incluindo ruído aleatório, sem gerar erros. Assim, se o seu programa pressupor uma codificação de 8 bits incorreta, ele decodificará lixo silenciosamente.



Caracteres sem sentido resultantes de decodificação errada são conhecidos como gremlins ou mojibake (文字化け – termo japonês para “texto transformado”).

O exemplo 4.7 mostra como o uso do codec incorreto pode gerar gremlins ou um `UnicodeDecodeError`.

Exemplo 4.7 – Decodificação de str para bytes: sucesso e tratamento de erro

```
>>> octets = b'Montr\xe9al' ❶
>>> octets.decode('cp1252') ❷
'Montréal'
>>> octets.decode('iso8859_7') ❸
```

```

'Montréal'
>>> octets.decode('koi8_r') ❶
'MontrMal'
>>> octets.decode('utf_8') ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace') ❸
'Montr❖al'

```

- ❶ Esses bytes são os caracteres de “Montréal” codificados como latin1; '\xe9' é o byte para “é”.
- ❷ Decodificar com 'cp1252' (Windows 1252) funciona, pois ele é um superconjunto próprio de latin1.
- ❸ ISO-8859-7 foi criado para grego, portanto o byte '\xe9' é interpretado incorretamente, mas nenhum erro é gerado.
- ❹ KOI8-R foi criado para russo. Agora '\xe9' corresponde à letra “И” do alfabeto cirílico.
- ❺ O codec 'utf_8' detecta que octets não é um UTF-8 válido e gera UnicodeDecodeError.
- ❻ Ao usar o tratamento de erro 'replace', '\xe9' é substituído por “❖” (código Unicode U+FFFD), o código REPLACEMENT CHARACTER oficial do Unicode cujo propósito é representar caracteres desconhecidos.

SyntaxError ao carregar módulos com codificação inesperada

UTF-8 é a codificação de fonte default para Python 3, assim como ASCII era o default em Python 2 (a partir da versão 2.5). Se você carregar um módulo .py que contenha dados que não sejam UTF-8 e não fizer nenhuma declaração de codificação, uma mensagem como esta será exibida:

```

SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line 1, but no
encoding declared; see http://python.org/dev/peps/pep-0263/ for details

```

Como UTF-8 é amplamente usado em sistemas GNU/Linux e OSX, um cenário provável é abrir um arquivo .py criado em Windows com cp1252. Observe que esse erro ocorre mesmo em Python para Windows, pois a codificação default para Python 3 é UTF-8 em todas as plataformas.

Para corrigir esse problema, adicione um comentário mágico coding no início do arquivo, como mostra o exemplo 4.8.

Exemplo 4.8 – ola.py: “Hello, World!” em português

```
# coding: cp1252  
print('Olá, Mundo!')
```



Agora que o código-fonte de Python 3 não está mais limitado a ASCII e tem a excelente codificação UTF-8 como default, a melhor “correção” para códigos-fonte em codificações legadas como 'cp1252' é convertê-los para UTF-8 e deixar de lado os comentários `coding`. Se seu editor não aceita UTF-8, é hora de trocá-lo.

Identificadores não-ASCII no código-fonte: deveria usá-los?

Python 3 permite usar identificadores que não são ASCII no código-fonte:

```
>>> ação = 'PBR'  
>>> ε = 10**-6 # ε = letra grega epsilon
```

Alguns não gostam dessa ideia. O argumento mais comum para ater-se aos identificadores ASCII é que isso facilita a leitura e a alteração do código para todos. Esse argumento ignora uma questão: você quer que seu código-fonte seja legível e editável pelo público-alvo desejado e esse grupo pode não ser formado por “todos”. Se o código pertence a uma corporação internacional ou é um código aberto e você quer ter colaboradores do mundo todo, os identificadores devem estar em inglês; nesse caso, tudo de que você precisa é de ASCII.

No entanto, se você é professor no Brasil, seus alunos acharão mais fácil ler um código que use variáveis e nomes de função em português, com uma ortografia correta. Eles não terão nenhuma dificuldade em digitar as cedilhas e as vogais acentuadas em seus teclados apropriados a essa localidade.

Agora que a linguagem Python é capaz de fazer parse de nomes Unicode e UTF-8 é a codificação-padrão de códigos-fonte, não vejo nenhum motivo para codificar os identificadores em português sem acentos, como costumávamos fazer em Python 2 por necessidade – a menos que seja necessário que seu código execute também em Python 2. Se os nomes estiverem em português, deixar de lado os acentos não deixará o código mais legível para ninguém.

Esse é o meu ponto de vista como brasileiro falante de português, porém acredito que isso se aplique além de fronteiras e culturas: escolha a língua que deixe o código mais fácil de ser lido pela equipe e então use os caracteres necessários para uma ortografia correta.

Suponha que você tenha um arquivo-texto, seja um código-fonte ou uma poesia, mas não saiba qual é a codificação. Como você poderá identificá-la? A próxima seção responderá a essa pergunta recomendando uma biblioteca.

Como descobrir a codificação de uma sequência de bytes

Como você pode descobrir a codificação de uma sequência de bytes? Resposta curta e grossa: não pode. Você deve ser informado.

Alguns protocolos de comunicação e formatos de arquivo, como HTTP e XML, contêm cabeçalhos que nos informam explicitamente como o conteúdo está codificado. Podemos ter certeza de que alguns streams de bytes não são ASCII porque eles contêm valores de bytes acima de 127, e o modo como UTF-8 e UTF-16 são construídos também limita as sequências possíveis de bytes. Mas, mesmo assim, jamais poderemos ter 100% de certeza de que um arquivo binário é ASCII ou UTF-8 somente porque alguns padrões de bits estão ausentes.

Contudo, considerando que as línguas humanas também têm suas regras e restrições, depois de presumir que um stream de bytes é um *texto puro* produzido por um ser humano, talvez seja possível detectar a sua codificação usando métodos heurísticos e estatísticos. Por exemplo, se bytes `b'\x00'` forem comuns, provavelmente a codificação será de 16 ou de 32 bits, e não um esquema de 8 bits, pois caracteres nulos em texto puro são bugs; se a sequência de bytes `b'\x20\x00'` aparecer com frequência, é provável que seja o caractere de espaço (U+0020) em uma codificação UTF-16LE, e não o caractere obscuro U+2000 EN QUAD – seja lá o que for isso.

É assim que o pacote *Chardet* — *The Universal Character Encoding Detector* (Chardet – o detector universal de codificação de caracteres, <https://pypi.python.org/pypi/chardet>) trabalha para identificar uma das 30 codificações aceitas. Chardet é uma biblioteca Python que você pode usar em seus programas, mas ela inclui também um utilitário `chardetect` de linha de comando. Eis o que ele informa sobre o arquivo-fonte deste capítulo:

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

Embora as sequências binárias de textos codificados geralmente não tenham pistas explícitas de sua codificação, os formatos UTF podem ter uma marca de ordem de byte prefixada ao conteúdo textual. Isso será explicado a seguir.

BOM: um gremlin útil

No exemplo 4.5, talvez você tenha notado alguns bytes extras no início de uma sequência codificada com UTF-16. Aqui está ela novamente:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfe\x00l\x00 \x00N\x00i\x00\x0f1\x00o\x00'
```

Os bytes são `b'\xff\xfe'`. Esse é o *BOM* – Byte-Order Mark (marca de ordem de byte) – que indica a ordem de bytes “*little-endian*” da CPU Intel em que a codificação foi realizada.

Em um computador *little-endian*, para cada código Unicode, o byte menos significativo vem antes: a letra 'E', cujo código Unicode é U+0045 (decimal 69), é codificada como 69 e 0 nos bytes com offsets 2 e 3:

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

Em uma CPU *big-endian*, a codificação será inversa; 'E' será codificado como 0 e 69.

Para evitar confusão, a codificação UTF-16 prefixa o texto a ser codificado com o caractere especial ZERO WIDTH NO-BREAK SPACE (U+FEFF), que é invisível. Em um sistema *little-endian*, isso é codificado como `b'\xff\xfe'` (decimais 255, 254). Uma vez que, propositalmente, não existe um caractere U+FFFE, a sequência de bytes `b'\xff\xfe'` deve significar ZERO WIDTH NO-BREAK SPACE em uma codificação *little-endian*, portanto o codec sabe qual será a ordem de bytes a ser usada.

Há uma variante de UTF-16 – UTF-16LE – que é *little-endian* por definição, e outra – UTF-16BE – que é *big-endian* por definição. Se você usá-las, o BOM não será gerado:

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

Se estiver presente, o BOM deverá ser filtrado pelo codec UTF-16, de modo que você obterá somente o conteúdo do texto propriamente dito do arquivo, sem ZERO WIDTH NO-BREAK SPACE na frente. O padrão afirma que, se um arquivo for UTF-16 e não tiver nenhum BOM, deve-se supor que seja UTF-16BE (*big-endian*). Entretanto a arquitetura Intel x86 é *little-endian*, portanto há muitos UTF-16 *little-endian* sem BOM por aí.

Esse problema todo de *endianness* afeta somente as codificações que usam palavras binárias de mais de um byte, como UTF-16 e UTF-32. Uma grande vantagem do UTF-8 é que ele gera a mesma sequência de bytes, independentemente do *endianness* do computador, portanto o BOM não será necessário. Apesar disso, alguns aplicativos Windows (com destaque para o Notepad) adicionam o BOM a arquivos UTF-8 de qualquer maneira – e o Excel depende do BOM para identificar um arquivo UTF-8; caso contrário, ele supõe que o conteúdo está codificado com uma página de código

(codepage) Windows. O caractere U+FEFF codificado em UTF-8 é a sequência de três bytes `b'\xef\xbb\xbf'`. Portanto, se um arquivo começar com esses três bytes, é provável que seja um arquivo UTF-8 com um BOM. No entanto a linguagem Python não presume automaticamente que um arquivo seja UTF-8 somente porque ele começa com `b'\xef\xbb\xbf'`.

Passaremos agora para o tratamento de arquivos-texto em Python 3.

Lidando com arquivos-texto

A melhor prática para lidar com textos é o “Sanduíche Unicode” (Figura 4.2).⁴ Isso quer dizer que bytes deve ser decodificado para str tão cedo quanto possível na entrada (por exemplo, ao abrir um arquivo para leitura). O “recheio” do sanduíche é a lógica de negócios de seu programa, em que o tratamento do texto é feito exclusivamente em objetos str. Você jamais deverá codificar ou decodificar no meio de outro processamento. Na saída, o str é codificado para bytes o mais tarde possível. A maioria dos frameworks web funciona dessa maneira, e raramente entramos em contato com bytes ao usar tais frameworks. Em Django, por exemplo, suas views devem gerar str Unicode; o próprio Django trata de codificar a resposta em bytes usando UTF-8 por padrão.

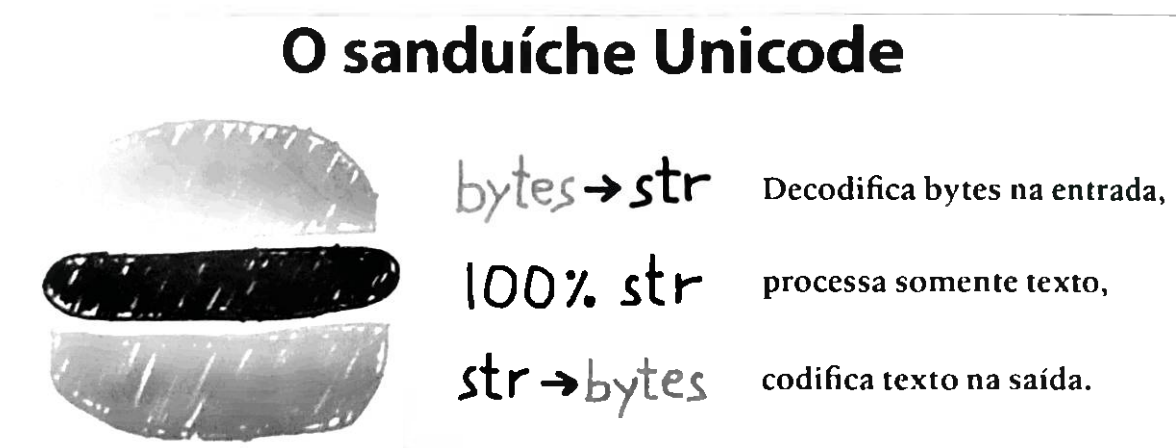


Figura 4.2 – Sanduíche Unicode: melhor prática para processamento de texto atualmente

Python 3 ajuda a seguir o conselho do sanduíche Unicode, pois a função embutida `open` faz a decodificação necessária ao ler e codifica ao gravar arquivos em modo texto, portanto tudo que você obtém de `my_file.read()` e passa para `my_file.write(text)` são objetos str.⁵

⁴ A primeira vez que vi o termo “Sanduíche Unicode” foi na excelente apresentação “Pragmatic Unicode” (Unicode pragmático) de Ned Batchelder (<http://nedbatchelder.com/text/unipain/unipain.html>) na US PyCon 2012.

⁵ Usuários de Python 2.6 ou 2.7 devem usar `io.open()` para ter decodificação/codificação automática ao ler/escrever.

Sendo assim, usar arquivos-texto é simples. No entanto, se você depender de codificações default, terá problemas.

Considere a sessão de console do exemplo 4.9. Você é capaz de identificar o bug?

Exemplo 4.9 – Um problema de codificação ligado à plataforma (se testar este código em seu computador, você poderá ou não ver o problema)

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

O bug: especifiquei a codificação UTF-8 ao gravar o arquivo, porém não fiz isso ao lê-lo, de modo que o Python pressupôs a codificação default do sistema – Windows 1252 – e os bytes no final do arquivo foram decodificados como os caracteres 'Ã©', e não como 'é'.

Executei o exemplo 4.9 em um computador com Windows 7. Os mesmos comandos executados em um GNU/Linux ou Mac OSX recentes funcionam perfeitamente porque a codificação default deles é UTF-8, o que dá a falsa impressão de que tudo está bem. Se o argumento de codificação fosse omitido quando o arquivo foi aberto para escrita, a codificação default da localidade teria sido usada e leríamos o arquivo corretamente usando a mesma codificação. Mas, nesse caso, esse script criaria arquivos com um conteúdo diferente em bytes de acordo com a plataforma ou, até mesmo, com diferentes configurações de localidade na mesma plataforma, criando problemas de compatibilidade.



Um código que precisa rodar em diversos computadores ou em várias ocasiões jamais deve depender de codificações default. Sempre passe um argumento `encoding=` explícito ao abrir arquivos-texto, pois o default poderá mudar de um computador para outro, ou de um dia para outro.

Um detalhe curioso no exemplo 4.9 é que a função `write` no primeiro comando informa que quatro caracteres foram escritos, porém, na próxima linha, cinco caracteres são lidos. O exemplo 4.10 apresenta uma versão estendida do exemplo 4.9, explicando esse fato, além de outros detalhes.

Exemplo 4.10 – Inspeção mais detalhada do exemplo 4.9 executando em Windows revela o bug e mostra como corrigi-lo

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café')
4 ❷
```



```

>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size
5 ❸
>>> fp2 = open('cafe.txt')
>>> fp2 ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ❺
'cp1252'
>>> fp2.read()
'cafÃ©' ❻
>>> fp3 = open('cafe.txt', encoding='utf_8') ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read()
'café' ❽
>>> fp4 = open('cafe.txt', 'rb') ❾
>>> fp4
<_io.BufferedReader name='cafe.txt'> ❿
>>> fp4.read() ⓫
b'caf\xc3\xa9'

```

- ❶ Por padrão, `open` opera em modo texto e devolve um objeto `TextIOWrapper`.
- ❷ O método `write` em um `TextIOWrapper` devolve o número de caracteres Unicode escritos.
- ❸ `os.stat` informa que o arquivo contém 5 bytes; UTF-8 codifica 'é' com 2 bytes: `0xc3` e `0xa9`.
- ❹ Abrir um arquivo-texto sem uma codificação explícita devolve um `TextIOWrapper` com a codificação definida com um default conforme a localidade.
- ❺ Um objeto `TextIOWrapper` tem um atributo de codificação que você pode inspecionar: `cp1252` nesse caso.
- ❻ Na codificação `cp1252` do Windows, o byte `0xc3` é um “Ã” (A com til) e `0xa9` é o símbolo de copyright.
- ❼ Abre o mesmo arquivo com a codificação correta.
- ❽ O resultado esperado: os mesmos quatro caracteres Unicode para 'café'.
- ❾ A flag `'rb'` abre um arquivo para leitura em modo binário.
- ❿ O objeto devolvido é um `BufferedReader`, e não um `TextIOWrapper`.
- ⓫ Leitura que retorna bytes, conforme esperado.



Não abra arquivos-texto em modo binário, a menos que você precise analisar o conteúdo do arquivo para determinar a codificação – mesmo assim, use Chardet em vez de reinventar a roda (veja a seção “Como descobrir a codificação de uma sequência de bytes” na página 142). Um código normal deve usar o modo binário somente para abrir arquivos binários, por exemplo, imagens raster.

O problema do exemplo 4.10 está no fato de depender de uma configuração default ao abrir um arquivo-texto. Esses defaults podem ter diversas origens, como mostra a próxima seção.

Defaults de codificação: um hospício

Várias configurações afetam a codificação default para I/O em Python. Veja o script *default_encodings.py* no exemplo 4.11.

Exemplo 4.11 – Explorando os defaults para codificação

```
import sys, locale

expressions = """
    locale.getpreferredencoding()
    type(my_file)
    my_file.encoding
    sys.stdout.isatty()
    sys.stdout.encoding
    sys.stdin.isatty()
    sys.stdin.encoding
    sys.stderr.isatty()
    sys.stderr.encoding
    sys.getdefaultencoding()
    sys.getfilesystemencoding()
    """

my_file = open('dummy', 'w')
for expression in expressions.split():
    value = eval(expression)
    print(expression.rjust(30), '->', repr(value))
```

A saída do exemplo 4.11 em GNU/Linux (Ubuntu 14.04) e em OSX (Mavericks 10.9) é idêntica e mostra que UTF-8 é usado em toda parte nesses sistemas:

```
$ python3 default_encodings.py
locale.getpreferredencoding() -> 'UTF-8'
type(my_file) -> <class '_io.TextIOWrapper'>
```

```

my_file.encoding -> 'UTF-8'
sys.stdout.isatty() -> True
sys.stdout.encoding -> 'UTF-8'
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'UTF-8'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'UTF-8'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'utf-8'

```

Em Windows, porém, a saída está no exemplo 4.12.

Exemplo 4.12 – Codificações default em cmd.exe no Windows 7 (SP 1) localizado para o Brasil; PowerShell apresenta o mesmo resultado.

```

Z:\>chcp ❶
Página de código ativa: 850
Z:\>python default_encodings.py ❷
locale.getpreferredencoding() -> 'cp1252' ❸
    type(my_file) -> <class '_io.TextIOWrapper'>
    my_file.encoding -> 'cp1252' ❹
sys.stdout.isatty() -> True ❺
sys.stdout.encoding -> 'cp850' ❻
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'cp850'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'cp850'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'mbcs'

```

- ❶ chcp mostra a página de código ativa para o console: 850.
- ❷ Executando *default_encodings.py* com saída para o console.
- ❸ `locale.getpreferredencoding()` é a configuração mais importante.
- ❹ Arquivos-texto usam `locale.getpreferredencoding()` por padrão.
- ❺ A saída está sendo enviada para o console, portanto `sys.stdout.isatty()` é True.
- ❻ Sendo assim, `sys.stdout.encoding` é igual à codificação do console.

Se a saída for redirecionada para um arquivo, desta maneira:

```
Z:\>python default_encodings.py > encodings.log
```

o valor de `sys.stdout.isatty()` se tornará False e `sys.stdout.encoding` será definido por `locale.getpreferredencoding()`, que é 'cp1252' nesse computador.

Observe que há quatro codificações diferentes no exemplo 4.12:

- Se você omitir o argumento `encoding` ao abrir um arquivo, o default será dado por `locale.getpreferredencoding()` ('cp1252' no exemplo 4.12).
- A codificação de `sys.stdout/stdin/stderr` é dada pela variável de ambiente `PYTHONIOENCODING` (<http://bit.ly/1IqvCUZ>) se ela estiver presente; caso contrário, será herdada do console ou definida por `locale.getpreferredencoding()` se a saída/entrada for redirecionada para/de um arquivo.
- `sys.getdefaultencoding()` é usado internamente por Python para converter dados binários para/de str; isso acontece com menos frequência em Python 3, mas continua ocorrendo.⁶ Não há suporte para alterar essa configuração.⁷
- `sys.getfilesystemencoding()` é usado para codificar/decodificar nomes de arquivo (e não os seus conteúdos). É usado quando `open()` recebe um argumento str para o nome do arquivo; se o nome do arquivo for especificado como um argumento bytes, ele será passado sem alteração à API do sistema operacional. O *Unicode HOWTO* de Python (<https://docs.python.org/3/howto/unicode.html>) afirma o seguinte: “em Windows, Python utiliza o nome `mbcs` para se referir à codificação configurada no momento, qualquer que seja ela”. O acrônimo MBCS quer dizer Multi Byte Character Set, que, para a Microsoft, são as codificações legadas de tamanho variável como `gb2312` ou `Shift_JIS`, mas não UTF-8. [Sobre esse assunto, uma resposta útil em StackOverflow está em “Difference between MBCS and UTF-8 on Windows” (Diferença entre MBCS e UTF_8 em Windows, <http://bit.ly/1IqvRPV>).]



Em GNU/Linux e em OSX, todas essas codificações são definidas como UTF-8 por padrão, e tem sido assim há vários anos, portanto o I/O trata todos os caracteres Unicode. Em Windows, não só codificações diferentes são usadas no mesmo sistema como também, geralmente, são páginas de código como 'cp850' ou 'cp1252', que aceitam somente ASCII com 127 caracteres adicionais, nem sempre iguais de uma codificação para outra. Desse modo, os usuários de Windows têm muito mais chances de se depararem com erros de codificação, a menos que tomem muito cuidado.

6 Enquanto pesquisava esse assunto, não encontrei uma lista de situações em que Python 3 convertesse bytes para str internamente. Antoine Pitrou, core developer de Python, afirma na lista `comp.python.devel` (<http://bit.ly/1IqvSU2>) que as funções internas de CPython que dependem dessas conversões “não têm muito uso em py3k”.

7 A função `sys.setdefaultencoding` de Python 2 era usada incorretamente e não está mais documentada em Python 3. Ela havia sido criada para ser usada pelos core developers quando a codificação default interna de Python ainda não tivesse sido decidida. Na mesma lista de discussão `comp.python.devel` (<http://bit.ly/1IqvN2J>), Marc-André Lemburg afirma que `sys.setdefaultencoding` jamais deve ser chamada pelo código de usuário e que os únicos valores aceitos por CPython são 'ascii' em Python 2 e 'utf-8' em Python 3.

Em suma, a configuração mais importante de codificação é aquela devolvida por `locale.getpreferredencoding()`: é o default para abrir arquivos-texto e para `sys.stdout/stdin/stderr` quando forem redirecionados para arquivos. No entanto, na documentação (<http://bit.ly/1IqvYLP>), podemos ver o seguinte (em parte):

```
locale.getpreferredencoding(do_setlocale=True)
```

Devolve a codificação usada para dados de texto de acordo com as preferências do usuário. As preferências do usuário são expressas de modo distinto em sistemas diferentes e podem não estar disponíveis por meio de programação em alguns sistemas, portanto essa função retorna somente um palpite. [...]

Sendo assim, o melhor conselho sobre os defaults de codificação é: não dependa deles.

Se seguir o conselho do sanduíche Unicode e sempre for explícito sobre as codificações em seus programas, você evitará muito sofrimento. Infelizmente, o Unicode pode causar sofrimento mesmo que você converta seus bytes corretamente para str. As duas próximas seções discutem assuntos que são simples na terra do ASCII, porém se tornam bem complexos no planeta Unicode: normalização (isto é, converter texto em uma representação uniforme para comparações) e ordenação de texto.

Normalizando Unicode para comparações mais seguras

As comparações de string são complicadas pelo fato de o Unicode ter caracteres combinados: diacríticos e outras marcas associadas ao caractere anterior, que aparecem como se fossem um só quando são exibidos.

Por exemplo, a palavra “café” pode ser composta de duas maneiras – usando quatro ou cinco códigos Unicode –, porém o resultado terá exatamente a mesma aparência:

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

O código Unicode U+0301 é o `COMBINING ACUTE ACCENT`. Usá-lo após “e” resulta em “é”. No padrão Unicode, sequências como ‘é’ e ‘e\u0301’ são chamadas de “equivalentes canônicos”, e supõe-se que as aplicações devam tratá-las como iguais. Porém o interpretador Python vê duas sequências diferentes de códigos Unicode e as considera diferentes.

A solução é usar a normalização de Unicode por meio da função `unicodedata.normalize`. O primeiro argumento dessa função é uma de quatro strings: 'NFC', 'NFD', 'NFKC' e 'NFKD'. Vamos começar pelas duas primeiras.

A normalização NFC (Normalization Form C) compõe os códigos Unicode de modo a gerar a menor string equivalente, enquanto NFD decompõe, expandindo os caracteres compostos em caracteres básicos, e separa os caracteres combinados. Essas duas normalizações fazem as comparações funcionarem conforme esperado:

```
>>> from unicodedata import normalize
>>> s1 = 'café' # "e" composto com acento agudo
>>> s2 = 'cafe\u0301' # "e" decomposto e acento agudo
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

Os teclados ocidentais normalmente geram caracteres compostos, portanto os textos digitados pelos usuários estarão em NFC por padrão. No entanto, por questões de segurança, talvez seja conveniente higienizar strings com `normalize('NFC', user_text)` antes de salvá-las. NFC também é a forma de normalização recomendada pelo W3C em *Character Model for the World Wide Web: String Matching and Searching* (Modelo de caracteres para a World Wide Web: correspondência e pesquisa de strings, <http://www.w3.org/TR/charmod-norm/>).

Alguns caracteres únicos são normalizados por NFC como outro caractere único. O símbolo da unidade de resistência elétrica ohm (Ω) é normalizado para a letra grega ômega maiúscula. Visualmente, elas são idênticas, porém não são comparadas como iguais, portanto é essencial normalizar para evitar surpresas:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
```

```
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

Nas siglas das duas outras formas de normalização – NFKC e NFKD – a letra K representa “compatibility” (compatibilidade). São formas mais fortes de normalização e afetam os chamados “caracteres de compatibilidade”. Embora um dos objetivos do Unicode seja ter um único código “canônico” para cada caractere, alguns caracteres aparecem mais de uma vez por questões de compatibilidade com padrões preexistentes. Por exemplo, o sinal de micro, 'μ' (U+00B5), foi adicionado ao Unicode para dar suporte à conversão de e para latin1, apesar de o mesmo caractere fazer parte do alfabeto grego com código Unicode U+03BC (GREEK SMALL LETTER MU). Desse modo, o sinal de micro é considerado um “caractere de compatibilidade”.

Nas formas NFKC e NFKD, cada caractere de compatibilidade é substituído por uma “decomposição de compatibilidade” de um ou mais caracteres considerados como uma representação “preferencial”, mesmo que haja alguma perda de formatação – o ideal é que a formatação seja responsabilidade de marcação externa, e não parte do Unicode. Para exemplificar, a decomposição de compatibilidade da fração que representa meio, '½' (U+00BD), é a sequência de três caracteres '1/2', e a decomposição de compatibilidade do sinal micro, 'μ' (U+00B5), é o mu minúsculo 'μ' (U+03BC).⁸

Veja como NFKC funciona na prática:

```
>>> from unicodedata import normalize, name
>>> half = '½'
>>> normalize('NFKC', half)
'1/2'
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'μ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('μ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

⁸ Curiosamente, o sinal de micro é considerado um “caractere de compatibilidade”, mas isso não ocorre com o símbolo de ohm. O resultado final é que NFC não afeta o sinal de micro, porém altera o símbolo de ohm para o ômega maiúsculo, enquanto NFKC e NFKD mudam tanto ohm quanto micro transformando-os em outros caracteres.

Embora '1/2' seja um substituto razoável para '½' e o sinal de micro seja, na verdade, a letra grega mu minúscula, converter '4²' para '42' altera o seu significado. Uma aplicação poderia armazenar '4²' como '4²', porém a função `normalize` não sabe nada sobre formatação. Sendo assim, NFKC ou NFKD podem perder ou distorcer informações, porém podem produzir representações intermediárias convenientes para pesquisa e indexação: os usuários podem ficar satisfeitos se uma pesquisa por '1/2 polegada' também encontrar documentos contendo '½ polegada'.



As normalizações NFKC e NFKD devem ser aplicadas com cuidado e somente em casos especiais – por exemplo, em pesquisa e indexação – e não para armazenagem permanente, pois essas transformações causam perda de dados.

Ao preparar textos para pesquisa ou indexação, outra operação conveniente é *case folding*, que é o nosso próximo assunto.

Case folding

Case folding é essencialmente a conversão de todo o texto em letras minúsculas, com algumas transformações adicionais. É realizada pelo método `str.casefold()` (novo em Python 3.3).

Para qualquer string `s` que contenha somente caracteres `latin1`, `s.casefold()`, produz o mesmo resultado que `s.lower()`, somente com duas exceções – o sinal de micro 'μ' é alterado para a letra grega mu minúscula (que tem a mesma aparência na maioria das fontes) e o Eszett ou “scharfes S” (ß) alemão torna-se “ss”:

```
>>> micro = 'μ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('μ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

Em Python 3.4, há 116 códigos Unicode para os quais `str.casefold()` e `str.lower()` devolvem resultados diferentes. Isso corresponde a 0,11 % de um total de 110.122 caracteres nomeados em Unicode 6.3.

Como ocorre normalmente com tudo que está relacionado a Unicode, o case folding é uma questão complicada, com muitos casos linguísticos especiais, porém os mantenedores de Python esforçaram-se para oferecer uma solução que deve funcionar para a maioria dos usuários.

Nas duas próximas seções, aplicaremos nosso conhecimento sobre normalização ao desenvolvimento de funções utilitárias.

Funções utilitárias para comparações normalizadas

Como vimos, NFC e NFD são seguras e permitem comparações sensatas entre strings Unicode. NFC é a melhor forma normalizada para a maioria das aplicações. `str.casefold()` é a solução para comparações que não levem em conta as diferenças entre letras maiúsculas e minúsculas (case-insensitive).

Se você trabalha com textos em diversas linguagens, um par de funções como `nfc_equal` e `fold_equal` do exemplo 4.13 serão acréscimos úteis à sua caixa de ferramentas.

Exemplo 4.13 – `normeq.py`: comparação de strings Unicode normalizadas

```
"""
```

Funções utilitárias para comparação de strings Unicode normalizadas.

Usando Normal Form C, considerando as diferenças entre letras minúsculas e maiúsculas (case sensitive):

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1 == s2
False
>>> nfc_equal(s1, s2)
True
>>> nfc_equal('A', 'a')
False
```

Usando Normal Form C com case folding:

```
>>> s3 = 'Straße'
>>> s4 = 'strasse'
>>> s3 == s4
False
>>> nfc_equal(s3, s4)
```

```
False
>>> fold_equal(s3, s4)
True
>>> fold_equal(s1, s2)
True
>>> fold_equal('A', 'a')
True
```

```
"""
```

```
from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

Indo além da normalização de Unicode e de case folding – ambos fazem parte do padrão Unicode –, às vezes faz sentido aplicar transformações mais profundas, por exemplo, mudar de 'café' para 'cafe'. Veremos quando e como isso se aplica na próxima seção.

“Normalização” extrema: removendo acentos

O molho secreto da pesquisa do Google envolve muitos truques, porém um deles aparentemente é ignorar diacríticos (por exemplo, acentos, cedilhas etc.), pelo menos em alguns contextos. Remover diacríticos não é uma forma 100% apropriada de normalização porque, frequentemente, altera o sentido das palavras e pode gerar falsos positivos na pesquisa. No entanto, ajuda a lidar com alguns fatos da vida: às vezes, as pessoas são preguiçosas ou ignorantes quanto ao uso correto de diacríticos, e as regras de ortografia mudam com o passar do tempo, o que significa que os acentos vêm e vão nas línguas vivas.

Afora a pesquisa, livrar-se dos diacríticos também deixa os URLs mais legíveis, pelo menos em idiomas baseados em latim. Dê uma olhada no URL do artigo da Wikipedia sobre a cidade de São Paulo:

```
http://en.wikipedia.org/wiki/%C3%A3o_Paulo
```

A parte %C3%A3 corresponde à letra única “ã” (“a” com til) apresentada em UTF-8 com escape para URL. A versão a seguir é muito mais simpática, mesmo que não apresente a ortografia correta:

```
http://en.wikipedia.org/wiki/Sao_Paulo
```

Para remover todos os diacríticos de uma `str`, você pode usar uma função como mostra o exemplo 4.14.

Exemplo 4.14 – Função para remover todas as marcas combinadas (módulo `sanitize.py`)

```
import unicodedata
import string

def shave_marks(txt):
    """Remove todas as marcas de diacríticos"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c)) ❷
    return unicodedata.normalize('NFC', shaved) ❸
```

- ❶ Decompõe todos os caracteres em caracteres-base e marcas combinadas.
- ❷ Filtra todas as marcas combinadas.
- ❸ Recompõe todos os caracteres.

O exemplo 4.15 mostra dois usos de `shave_marks`.

Exemplo 4.15 – Dois exemplos que usam `shave_marks` do exemplo 4.14

```
>>> order = '"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of açaí."'
>>> shave_marks(order)
'"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of acai."' ❶
>>> Greek = 'Ζέφυρος, Ζέφиро'
>>> shave_marks(Greek)
'Ζεφυρος, Zefiro' ❷
```

- ❶ Somente as letras “ë”, “ç” e “ï” foram substituídas.
- ❷ Tanto “É” quanto “é” foram substituídas.

A função `shave_marks` do exemplo 4.14 funciona bem, mas talvez vá longe demais. Geralmente, o motivo para remover diacríticos é alterar um texto latino para ASCII puro, porém `shave_marks` também altera caracteres não latinos – como as letras gregas – que jamais se tornarão ASCII somente por perderem seus acentos. Portanto faz sentido analisar cada caractere-base e remover as marcas associadas somente se o caractere-base for uma letra do alfabeto latino. É isso que o exemplo 4.16 faz.

Exemplo 4.16 – Função para remover marcas combinadas de caracteres latinos (instruções de importação foram omitidas, pois este código faz parte do módulo `sanitize.py` do exemplo 4.14)

```
def shave_marks_latin(txt):
    """Remove todas as marcas de diacríticos dos caracteres-base latinos"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    latin_base = False
    keepers = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base: ❷
            continue # ignora diacríticos em caracteres-base latinos
        keepers.append(c) ❸
        # se não é um caractere combinado, é um novo caractere-base
        if not unicodedata.combining(c): ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(keepers)
    return unicodedata.normalize('NFC', shaved) ❺
```

- ❶ Decompõe todos os caracteres em caracteres-base e marcas combinadas.
- ❷ Pula as marcas combinadas quando o caractere-base for latino.
- ❸ Caso contrário, mantém o caractere atual.
- ❹ Detecta o novo caractere-base e determina se é latino.
- ❺ Recompõe todos os caracteres.

Um passo mais radical seria substituir símbolos comuns em textos ocidentais (por exemplo, aspas curvas, travessões longos, bullets etc.) em equivalentes ASCII. É isso que a função `asciize` faz no exemplo 4.17.

Exemplo 4.17 – Transforma alguns símbolos tipográficos ocidentais em ASCII (este trecho de código também faz parte do módulo `sanitize.py` do exemplo 4.14)

```
single_map = str.maketrans("","f„†^<'\""•-->""", ❶
                           """"f"*^<'\""---~>""")

multi_map = str.maketrans({ ❷
    '€': '<euro>',
    '…': '...',
    'OE': 'OE',
    '™': '(TM)',
    'oe': 'oe',
    '‰': '<per mille>',
    '‡': '**',
})
```

```

multi_map.update(single_map) ❸

def dewinize(txt):
    """Substitui símbolos Win1252 por caracteres ou sequências ASCII"""
    return txt.translate(multi_map) ❹

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt)) ❺
    no_marks = no_marks.replace('ß', 'ss') ❻
    return unicodedata.normalize('NFKC', no_marks) ❼

```

- ❶ Cria tabela de mapeamento para substituição de caractere para caractere.
- ❷ Cria tabela de mapeamento para substituição de caractere para string.
- ❸ Combina as tabelas de mapeamento.
- ❹ dewinize não afeta texto ASCII ou latin1, somente os acréscimos da Microsoft ao latin1 em cp1252.
- ❺ Aplica dewinize e remove marcas de diacríticos.
- ❻ Substitui Eszett por “ss” (não estamos usando case fold nesse caso, pois queremos preservar a diferença entre letras maiúsculas e minúsculas).
- ❼ Aplica a normalização NFKC para compor caracteres com seus códigos de compatibilidade Unicode.

O exemplo 4.18 mostra asciize em uso.

Exemplo 4.18 – Dois exemplos que usam asciize do exemplo 4.17

```

>>> order = '"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of açai."'
>>> dewinize(order)
'"Herr Voß: - ½ cup of OEtker(TM) caffè latte - bowl of açai."' ❶
>>> asciize(order)
'"Herr Voss: - 1/2 cup of OEtker(TM) caffè latte - bowl of acai."' ❷

```

- ❶ dewinize substitui aspas curvas, bullets e ™ (símbolo de marca registrada).
- ❷ asciize aplica dewinize, remove diacríticos e substitui 'ß'.



Línguas diferentes têm suas próprias regras para remover diacríticos. Por exemplo, os alemães mudam 'ü' para 'ue'. Nossa função asciize não é tão sofisticada assim, portanto ela pode ou não ser adequada à sua língua. Contudo, para português, ela é razoável.

Em suma, as funções em *sanitize.py* vão muito além da normalização-padrão e realizam uma cirurgia profunda no texto, com boa chance de alterar o seu significado.

Só você pode decidir se deve ir tão longe conhecendo a língua-alvo, seus usuários e como o texto transformado será usado.

Com isso, concluímos nossa discussão sobre normalização de texto Unicode.

O próximo assunto relacionado a Unicode a ser desvendado é ordenação.

Ordenação de texto Unicode

A linguagem Python ordena sequências de qualquer tipo comparando os itens de cada sequência, um a um. Para strings, isso significa comparar os códigos Unicode. Infelizmente, isso gera resultados inaceitáveis para qualquer pessoa que use caracteres não-ASCII.

Considere a ordenação de uma lista de frutas cultivadas no Brasil:

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açai', 'caju', 'cajá']
```

As regras de ordenação variam para diferentes localidades, porém, em português e em diversas línguas que usem o alfabeto latino, os acentos e as cedilhas raramente fazem diferença na ordenação.⁹ Desse modo, “cajá” é ordenado como “caja” e deve vir antes de “caju”.

A lista `fruits` ordenada deve ser:

```
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

A maneira-padrão de ordenar textos que não sejam ASCII em Python é por meio da função `locale.strxfrm`, que, de acordo com a documentação do módulo `locale`, “transforma uma string em outra para ser usada em comparações que levem em conta a localidade (*locale*)”.

Para habilitar `locale.strxfrm`, inicialmente você precisa definir uma localidade adequada para a sua aplicação e rezar para que o sistema operacional tenha suporte para ela. Em GNU/Linux (Ubuntu 14.04), com a localidade `pt_BR`, a sequência de comandos do exemplo 4.19 funciona.

Exemplo 4.19 – Usando a função `locale.strxfrm` como chave de ordenação

```
>>> import locale
>>> locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
'pt_BR.UTF-8'
```

⁹ Os diacríticos afetam a ordenação somente nos raros casos em que eles sejam a única diferença entre duas palavras – nesse caso, a palavra com um diacrítico será ordenada após a palavra sem ele.


```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted_fruits = sorted(fruits, key=locale.strxfrm)
>>> sorted_fruits
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

Você precisa chamar `setlocale(LC_COLLATE, «sua localidade»)` antes de usar `locale.strxfrm` como chave ao ordenar.

Contudo há algumas ressalvas:

- Como as configurações de *locale* são globais, chamar `setlocale` em uma biblioteca não é recomendável. Sua aplicação ou o framework deve definir o *locale* quando o processo iniciar e não deverá alterá-la depois.
- O *locale* deve estar instalado no sistema operacional; se não estiver, `setlocale` levantará uma exceção `locale.Error: unsupported locale setting`.
- Você precisa saber como identificar o *locale*. Os identificadores são bem padronizados como `'language_code.encoding'` nos derivados de Unix, porém, em Windows, a sintaxe é mais complicada: `Language Name-Language Variant_Region Name.codepage`. Observe que `Language Name` (nome do idioma), `Language Variant` (variante do idioma) e `Region Name` (nome da região) podem conter espaços, porém as partes após a primeira são prefixadas com caracteres especiais diferentes: um hífen, um underline e um ponto. Todas as partes parecem ser opcionais, exceto o nome do idioma. Por exemplo: `English_United States.850` quer dizer que o nome do idioma é “English” (inglês), a região é “United States” (Estados Unidos) e a página de código é “850”. Os nomes de idiomas e de regiões compreendidos pelo Windows estão listados no artigo *Language Identifier Constants and Strings* do MSDN (Constantes e strings para identificação de idiomas, <http://bit.ly/1IqyKA1>), enquanto *Code Page Identifiers* (Identificadores de páginas de código, <http://bit.ly/1IqyP79>) lista os números para a última parte.¹⁰
- O *locale* deve estar corretamente implementado pelos fabricantes do sistema operacional. Tive sucesso em Ubuntu 14.04, mas não em OSX (Mavericks 10.9). Em dois Macs diferentes, a chamada a `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` devolveu a string `'pt_BR.UTF-8'` sem reclamar. Entretanto `sorted(fruits, key=locale.strxfrm)` gerou o mesmo resultado incorreto de `sorted(fruits)`. Também testei as localidades `fr_FR`, `es_ES` e `de_DE` em OSX, porém `locale.strxfrm` não fez o seu trabalho.¹¹

¹⁰ Agradeço a Leonardo Rachael, que foi além de suas obrigações como revisor técnico e pesquisou esses detalhes em Windows, apesar de, ele próprio, ser usuário de GNU/Linux.

¹¹ Novamente, não pude encontrar uma solução, mas encontrei outras pessoas que relataram o mesmo problema. Alex Martelli, um dos revisores técnicos, não teve nenhum problema em usar `setlocale` e `locale.strxfrm` em seu Mac com OSX 10.9. Em suma, seus resultados podem variar conforme a sua situação.

Desse modo, a solução da biblioteca-padrão para ordenação internacionalizada funciona, mas parece ter um bom suporte somente em GNU/Linux (talvez também em Windows, se você for um expert). Mesmo assim, depende das configurações de *locale*, o que é motivo para dores de cabeça na implantação de sistemas em produção. Felizmente, há uma solução mais simples: a biblioteca PyUCA, disponível em *PyPI*.

Ordenação com o Unicode Collation Algorithm

James Tauber, colaborador bastante produtivo de Django, deve ter sofrido com esse problema e criou a PyUCA (<https://pypi.python.org/pypi/pyuca/>) – uma implementação puramente Python do Unicode Collation Algorithm (UCA). O exemplo 4.20 mostra como é fácil usá-la.

Exemplo 4.20 – Usando o método `pyuca.Collator.sort_key`

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

É um código amigável e simplesmente funciona. Testei-o em GNU/Linux, OSX e Windows. Há suporte somente para Python 3.X no momento.

PyUCA não leva o *locale* em consideração. Se você precisa personalizar a ordenação, poderá fornecer o path para uma tabela personalizada de *collation* ao construtor `Collator()`. Sem configuração específica, ela usa `allkeys.txt` (<https://github.com/jtauber/pyuca>), que vem incluído no projeto. É somente uma cópia de *Default Unicode Collation Element Table* (Tabela default de elementos de Unicode Collation, <http://bit.ly/1IqAk54>) de Unicode 6.3.0.

A propósito, essa tabela é uma das várias que compõem a base de dados Unicode, que será o nosso próximo assunto.

Base de dados Unicode

O padrão Unicode oferece uma base de dados completa – na forma de diversos arquivos-texto estruturados – que inclui não só a tabela que mapeia códigos Unicode aos nomes dos caracteres como também os metadados sobre os caracteres individuais e como eles estão relacionados. Por exemplo, a base de dados Unicode registra se

um caractere pode ser exibido, se é uma letra, um dígito decimal ou outro símbolo numérico. É assim que os métodos `isidentifier`, `isprintable`, `isdecimal` e `isnumeric` de `str` funcionam; `str.casefold` também usa informações de uma tabela de Unicode.

O módulo `unicodedata` tem funções que devolvem metadados de caracteres; por exemplo, o nome oficial do caractere no padrão, se é um caractere que pode ser combinado com outro (por exemplo, um diacrítico como o til), e o valor numérico do símbolo para os seres humanos (não o seu código Unicode). O exemplo 4.21 mostra o uso de `unicodedata.name()` e de `unicodedata.numeric()`, juntamente com os métodos `.isdecimal()` e `.isnumeric()` de `str`.

Exemplo 4.21 – Demo dos metadados de caracteres numéricos da base de dados Unicode (os comentários numerados descrevem cada coluna da saída)

```
import unicodedata
import re

re_digit = re.compile(r'\d')

sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print('U+%04x' % ord(char),           ❶
          char.center(6),                 ❷
          're_digit' if re_digit.match(char) else '-', ❸
          'isdig' if char.isdigit() else '-',         ❹
          'isnum' if char.isnumeric() else '-',        ❺
          format(unicodedata.numeric(char), '5.2f'),  ❻
          unicodedata.name(char),                 ❼
          sep='\t')
```

- ❶ Código Unicode no formato U+0000.
- ❷ Caractere centralizado em uma str de tamanho 6.
- ❸ Mostra `re_digit` se o caractere corresponde à regex `r'\d'`.
- ❹ Mostra `isdig` se `char.isdigit()` for True.
- ❺ Mostra `isnum` se `char.isnumeric()` for True.
- ❻ Valor numérico formatado com largura 5 e duas casas decimais.
- ❼ Nome do caractere Unicode.

A execução do exemplo 4.21 fornece o resultado mostrado na figura 4.3.

```
$ python3 numerics_demo.py
U+0031  1    re_dig isdig  isnum  1.00  DIGIT ONE
U+00bc  ¼    -      -      isnum  0.25  VULGAR FRACTION ONE QUARTER
U+00b2  ²    -      isdig  isnum  2.00  SUPERScript TWO
U+0969  ३    re_dig isdig  isnum  3.00  DEVANAGARI DIGIT THREE
U+136b  ፫    -      isdig  isnum  3.00  ETHIOPIC DIGIT THREE
U+216b  Ⅻ    -      -      isnum  12.00 ROMAN NUMERAL TWELVE
U+2466  ⑦    -      isdig  isnum  7.00  CIRCLED DIGIT SEVEN
U+2480  ⑬    -      -      isnum  13.00 PARENTHEZIZED NUMBER THIRTEEN
U+3285  ⑤    -      -      isnum  6.00  CIRCLED IDEOGRAPH SIX
$
```

Figura 4.3 – Nove caracteres numéricos e seus metadados; *re_dig* quer dizer que o caractere corresponde à expressão regular *r^d*;

A sexta coluna da figura 4.3 é o resultado da chamada a `unicodedata.numeric(char)` no caractere. Ela mostra que o Unicode conhece o valor numérico dos símbolos que representam números. Portanto, se você quiser criar um aplicativo de planilha que aceite dígitos em tâmil ou algarismos romanos, vá em frente!

A figura 4.3 mostra que a expressão regular *r'^d'* corresponde ao dígito “1” e ao dígito devanágari 3, mas não a outros caracteres considerados dígitos pela função `isdigit`. O módulo `re` não é tão conhecedor de Unicode como poderia ser. O novo módulo `regex` disponível em PyPI foi projetado para tomar o lugar de `re` em algum momento e oferece um melhor suporte ao Unicode.¹² Retornaremos ao módulo `re` na próxima seção.

Ao longo deste capítulo, usamos diversas funções de `unicodedata`, mas há muitas outras que não discutimos. Consulte a documentação da biblioteca-padrão referente ao módulo `unicodedata` (<https://docs.python.org/3/library/unicodedata.html>).

Concluiremos o nosso passeio por *str* versus *bytes* dando uma olhada rapidamente em uma nova tendência: APIs de modo dual com funções que aceitam argumentos *str* ou *bytes*, com tratamento especial de acordo com o tipo.

APIs de modo dual para *str* e *bytes*

A biblioteca-padrão tem funções que aceitam argumentos *str* e *bytes* e se comportam de modo diferente de acordo com o tipo. Alguns exemplos estão nos módulos `re` e `os`.

str versus *bytes* em expressões regulares

Se você criar uma expressão regular com *bytes*, padrões como `\d` e `\w` corresponderão somente a caracteres ASCII; em comparação, se esses padrões forem especificados

¹² Embora não tenha sido melhor que `re` em identificar dígitos nesse exemplo em particular.

com str, eles corresponderão a dígitos ou letras Unicode além dos caracteres ASCII. O exemplo 4.22 e a figura 4.4 comparam a correspondência de letras, dígitos ASCII, sobrescritos e dígitos em tâmil com padrões str e bytes.

Exemplo 4.22 – ramanujan.py: compare o comportamento de expressões regulares str e bytes simples

```
import re

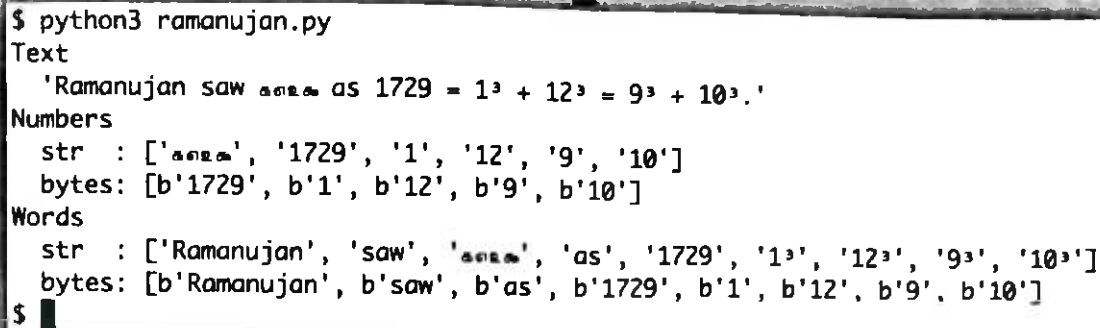
re_numbers_str = re.compile(r'\d+') ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+') ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef" ❸
            " as 1729 = 1³ + 12³ = 9³ + 10³.") ❹

text_bytes = text_str.encode('utf_8') ❺

print('Text', repr(text_str), sep='\n ')
print('Numbers')
print(' str :', re_numbers_str.findall(text_str)) ❻
print(' bytes:', re_numbers_bytes.findall(text_bytes)) ❼
print('Words')
print(' str :', re_words_str.findall(text_str)) ❽
print(' bytes:', re_words_bytes.findall(text_bytes)) ❾
```

- ❶ As duas primeiras expressões regulares são do tipo str.
- ❷ As duas últimas são do tipo bytes.
- ❸ Texto Unicode a ser pesquisado, contendo os dígitos em tâmil para 1729 (a linha lógica continua até o parêntese direito).
- ❹ Essa string é unida à anterior em tempo de compilação [veja a seção “2.4.2. String literal concatenation” (Concatenação de strings literais, <http://bit.ly/1IqE2vH>) em *The Python Language Reference* (Guia de referência da linguagem Python)].
- ❺ Uma string bytes é necessária para pesquisar com expressões regulares do tipo bytes.
- ❻ O padrão str `r'\d+'` corresponde aos dígitos em tâmil e ASCII.
- ❼ O padrão bytes `rb'\d+'` corresponde somente aos bytes ASCII para dígitos.
- ❽ O padrão str `r'\w+'` corresponde a letras, sobrescritos, tâmil e dígitos ASCII.
- ❾ O padrão bytes `rb'\w+'` corresponde somente aos bytes ASCII para letras e dígitos.



```
$ python3 ramanujan.py
Text
'Ramanujan saw as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
str : ['¹', '²', '³', '¹', '²', '³', '¹', '²', '³']
bytes: [b'¹', b'²', b'³', b'¹', b'²', b'³', b'¹', b'²', b'³']
Words
str : ['Ramanujan', 'saw', 'as', '1729', '1³', '12³', '9³', '10³']
bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'¹', b'²', b'³', b'¹', b'²', b'³']
$
```

Figura 4.4 – Captura de tela da execução de `ramanujan.py` do exemplo 4.22.

O exemplo 4.22 é trivial e foi criado para chamar a atenção para uma questão: você pode usar expressões regulares em `str` e em `bytes`, mas, no segundo caso, os bytes fora do intervalo ASCII serão tratados como caracteres não-dígito e não-palavra.

Em expressões regulares do tipo `str`, há uma flag `re.ASCII` que faz `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` e `\S` corresponderem somente a ASCII. Consulte a documentação do módulo `re` (<https://docs.python.org/3/library/re.html>) para ver os detalhes completos.

Outro módulo importante de modo dual é o `os`.

str versus bytes em funções de os

O kernel do GNU/Linux não foi projetado para lidar com Unicode, portanto, no mundo real, você poderá encontrar nomes de arquivo compostos de sequências de bytes que não são válidas em nenhum esquema sensato de codificação e não poderão ser codificadas para `str`. Servidores de arquivos com clientes que usam uma variedade de sistemas operacionais são particularmente suscetíveis a esse problema.

Para contornar esse obstáculo, todas as funções do módulo `os` que aceitam nomes de arquivo ou paths aceitam argumentos como `str` ou como `bytes`. Se uma função desse tipo for chamada com um argumento `str`, esse argumento será automaticamente convertido usando o codec nomeado por `sys.getfilesystemencoding()`, e a resposta do sistema operacional será decodificada com o mesmo codec. Quase sempre é isso que você vai querer, e estará de acordo com a boa prática do sanduíche Unicode.

Porém, se você precisa lidar com (ou talvez corrigir) nomes de arquivo que não possam ser tratados dessa maneira, poderá passar argumentos `bytes` às funções de `os` para obter valores de retorno em `bytes`. Esse recurso permite que você lide com qualquer nome ou path de arquivo, independentemente de quantos gremlins forem encontrados. Veja o exemplo 4.23.

Exemplo 4.23 – `listdir` com argumentos `str` e `bytes` e os resultados

```
>>> os.listdir('.') # ❶  
['abc.txt', 'digits-of-π.txt']  
>>> os.listdir(b'.') # ❷  
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

- ❶ O nome do segundo arquivo é “digits-of- π .txt” (com a letra grega π).
- ❷ Dado um argumento `byte`, `listdir` devolve os nomes de arquivo como `bytes`: `b'\xcf\x80'` é a codificação UTF-8 da letra grega π).

Para ajudar no tratamento manual de sequências `str` ou `bytes` que sejam nomes ou paths de arquivo, o módulo `os` oferece funções especiais de codificação e decodificação:

`fsencode(filename)`

Codifica `filename` (pode ser `str` ou `bytes`) para `bytes` usando o codec nomeado por `sys.getfilesystemencoding()` se `filename` for do tipo `str`; caso contrário, devolve o `filename` do tipo `bytes` inalterado.

`fsdecode(filename)`

Decodifica `filename` (pode ser `str` ou `bytes`) para `str` usando o codec nomeado por `sys.getfilesystemencoding()` se `filename` for do tipo `bytes`; caso contrário, devolve o `filename` do tipo `str` inalterado.

Em plataformas derivadas de Unix, essas funções usam o handler de erro `surrogateescape` (veja a caixa de texto a seguir) para evitar problemas com `bytes` inesperados. Em Windows, o handler de erro `strict` é usado.

Usando `surrogateescape` para lidar com gremlins

Um truque para lidar com `bytes` inesperados ou codificações desconhecidas é o handler de erro de codec `surrogateescape`, descrito na *PEP 383 – Non-decodable Bytes in System Character Interfaces* (Bytes que não podem ser decodificados em interfaces de caracteres do sistema, <https://www.python.org/dev/peps/pep-0383/>) introduzido em Python 3.1.

A ideia desse handler de erro é substituir cada `byte` que não puder ser decodificado por um código Unicode no intervalo de `U+DC00` a `U+DCFF` na chamada “Low Surrogate Area” (Área baixa dos substitutos) do padrão – um espaço de código sem caracteres atribuídos, reservado para uso interno das aplicações. Na codificação, esses códigos Unicode são convertidos de volta para os valores em `byte` que eles substituíram. Veja o exemplo 4.24.

Exemplo 4.24 – Usando o handler de erro `surrogateescape`

```
>>> os.listdir('.') ❶  
['abc.txt', 'digits-of-n.txt']  
>>> os.listdir(b'.') ❷  
[b'abc.txt', b'digits-of-\xcf\x80.txt']  
>>> pi_name_bytes = os.listdir(b'.')[1] ❸  
>>> pi_name_str = pi_name_bytes.decode('ascii', 'surrogateescape') ❹  
>>> pi_name_str ❺  
'digits-of-\udccf\udc80.txt'  
>>> pi_name_str.encode('ascii', 'surrogateescape') ❻  
b'digits-of-\xcf\x80.txt'
```

- ❶ Lista diretório com um nome de arquivo que não usa ASCII.
 - ❷ Vamos fingir que não sabemos qual é a codificação e obter os nomes dos arquivos como bytes.
 - ❸ `pi_names_bytes` é o nome do arquivo com o caractere `pi`.
 - ❹ Decodifica-o para `str` usando o codec `'ascii'` com `'surrogateescape'`.
 - ❺ Cada byte que não é ASCII é substituído por um código Unicode substituto (surrogate): `'\xcf\x80'` torna-se `'\udccf\udc80'`.
 - ❻ Codifica de volta para bytes ASCII: cada código Unicode substituto é trocado pelo byte que ele substituiu.
-

Com isso, encerramos a nossa exploração de `str` e bytes. Se você ainda está comigo, parabéns!

Resumo do capítulo

Iniciamos o capítulo refutando a noção de `1 character == 1 byte`. À medida que o mundo adota o Unicode (80% dos sites já usam UTF-8), devemos manter o conceito de strings de texto separado das sequências binárias que as representam em arquivos, e Python 3 reforça essa separação.

Após uma rápida visão geral dos tipos para sequências binárias – `bytes`, `bytearray` e `memoryview` –, passamos para a codificação e a decodificação, com uma amostra de codecs importantes, seguida de abordagens para evitar ou lidar com os infames `UnicodeEncodeError`, `UnicodeDecodeError` e `SyntaxError` causados por uma codificação incorreta em arquivos-fontes Python.

Durante a discussão sobre código-fonte, apresentei minha posição no debate sobre identificadores não-ASCII: se os mantenedores da base de código quiserem usar uma língua com caracteres que não sejam ASCII, os identificadores devem seguir essa linha – a menos que o código deva executar também em Python 2. Porém, se o projeto pretende atrair uma base de colaboradores internacionais, os identificadores devem ser compostos de palavras em inglês, e, nesse caso, ASCII será suficiente.

Em seguida, consideramos a teoria e a prática da detecção de codificação na ausência de metadados: em teoria, essa detecção não pode ser feita, porém, na prática, o pacote Chardet é capaz de extrair adequadamente a codificação para diversas codificações populares. As marcas de ordem de bytes foram então apresentadas como a única pista de codificação encontrada em arquivos UTF-16 e UTF-32 – às vezes, em arquivos UTF-8 também.

Na seção seguinte, mostramos como abrir arquivos-texto, que é uma tarefa fácil, exceto por uma armadilha: o argumento nomeado `encoding=` não é obrigatório quando você abre um arquivo-texto, embora devesse ser. Se você deixar de especificar a codificação, acabará com um programa que conseguirá gerar arquivos de “texto puro” incompatíveis entre plataformas diferentes devido a codificações default conflitantes. Em seguida, expusemos as diferentes configurações de codificação usadas por Python como default e mostramos como identificá-las: `locale.getpreferredencoding()`, `sys.getfilesystemencoding()`, `sys.getdefaultencoding()` e as codificações para os arquivos de I/O padrão (por exemplo, `sys.stdout.encoding`). Uma triste constatação para os usuários de Windows é que essas configurações geralmente têm valores distintos no mesmo computador e os valores são mutuamente incompatíveis; os usuários de GNU/Linux e de OSX, em comparação, estão em uma situação mais feliz, pois UTF-8 é default praticamente em toda parte.

As comparações de texto são surpreendentemente complicadas porque o Unicode oferece diversas maneiras de representar alguns caracteres, portanto a normalização é um pré-requisito para comparações entre textos. Além de explicar a normalização e o *case folding*, apresentamos algumas funções utilitárias que você poderá adaptar de acordo com suas necessidades, incluindo transformações drásticas como remover todos os acentos. Vimos então como ordenar corretamente um texto Unicode aproveitando o módulo `locale` padrão – com algumas ressalvas – e uma alternativa que não depende de configurações complicadas de `locale`: o pacote externo PyUCA.

Por fim, demos uma olhada na base de dados Unicode (uma fonte de metadados sobre todos os caracteres) e concluímos com uma rápida discussão sobre APIs de modo dual (por exemplo, os módulos `re` e `os`, em que algumas funções podem ser chamadas com argumentos `str` ou `bytes`, fornecendo resultados diferentes, porém adequados).

Leituras complementares

A apresentação “Pragmatic Unicode – or – How Do I Stop the Pain?” (Unicode pragmático – ou – como parar de sofrer?, <http://nedbatchelder.com/text/unipain.html>) de Ned Batchelder na PyCon US 2012 foi excelente. Ned é tão profissional que publicou uma transcrição completa da palestra, juntamente com os slides e o vídeo. Esther Nam e Travis Fischer fizeram uma excelente apresentação na PyCon 2014, “Character encoding and Unicode in Python: How to (𐀀 𐀁) 𐀂 with dignity” (Codificação de caracteres e Unicode em Python: como (𐀀 𐀁) 𐀂 com dignidade, slides em <http://bit.ly/1JzF1MY>, vídeo em <http://bit.ly/1JzF37P>), da qual extraí a curta e agradável epígrafe deste capítulo: “Seres humanos usam texto. Computadores falam bytes.” Lennart Regebro – um dos revisores técnicos deste livro – apresenta seu “Useful Mental Model of Unicode (UMMU)” (Modelo mental útil de Unicode) no pequeno post “Unconfusing Unicode: What Is Unicode?” (Descomplicando o Unicode: o que é Unicode?, <https://regebro.wordpress.com/2011/03/23/unconfusing-unicode-what-is-unicode/>). O Unicode é um padrão complexo e o UMMU de Lennart é realmente um ponto de partida útil.

O Unicode HOWTO oficial (<https://docs.python.org/3/howto/unicode.html>) na documentação de Python aborda o assunto de vários ângulos diferentes, desde uma boa introdução histórica até detalhes de sintaxe, codecs, expressões regulares, nomes de arquivo e as melhores práticas para I/O que leve o Unicode em consideração (ou seja, o sanduíche Unicode), com uma variedade de links adicionais para referência em cada seção. O capítulo 4 “Strings” (<http://www.diveintopython3.net/strings.html>) do sensacional livro *Dive into Python 3* (<http://www.diveintopython3>) de Mark Pilgrim também oferece uma introdução muito boa ao suporte de Python 3 ao Unicode. No mesmo livro, o capítulo 15 (<http://bit.ly/1IqJ63d>) descreve como a biblioteca Chardet foi portada de Python 2 para Python 3 – um caso de estudo de muito valor, considerando que a mudança do tipo `str` antigo para o novo `bytes` é a causa do sofrimento na maior parte das migrações e que essa é uma preocupação principal em uma biblioteca projetada para identificar codificações.

Se você conhece Python 2, mas Python 3 é novidade, *What’s New in Python 3.0* (O que há de novo em Python 3.0, <http://bit.ly/1IqJ8YH>) de Guido van Rossum tem quinze itens listados que resumem o que mudou, com vários links. Guido começa com a seguinte afirmação brusca: “Tudo que você achou que sabia sobre dados binários e Unicode mudou.” O post de blog “The Updated Guide to Unicode on Python” (O guia atualizado para Unicode em Python, <http://bit.ly/1IqJcrD>) de Armin Ronacher é detalhado e destaca algumas das armadilhas de Unicode em Python 3 (Armin não é um grande fã de Python 3).

O capítulo 2 “Strings and Text” do livro *Python Cookbook*, 3ª edição (O’Reilly)¹³ de David Beazley e Brian K. Jones contém diversas receitas que lidam com normalização de Unicode, higienização de textos e execução de operações orientadas a textos em seqüências de bytes. O capítulo 5 discute arquivos e I/O e inclui “Recipe 5.17. Writing Bytes to a Text File” (Escrever bytes em um arquivo-texto), mostrando que, subjacente a qualquer arquivo-texto, há sempre um stream binário que pode ser acessado diretamente quando for necessário. Mais adiante no Cookbook, o módulo `struct` é colocado em uso na “Recipe 6.11. Reading and Writing Binary Arrays of Structures” (Ler e escrever arrays de estruturas binárias).

O blog Python Notes de Nick Coghlan tem dois posts muito relevantes referentes a este capítulo: “Python 3 and ASCII Compatible Binary Protocols” (Python 3 e protocolos binários compatíveis com ASCII, <http://bit.ly/1dYuNJa>) e “Processing Text Files in Python 3” (Processando arquivos-texto em Python 3, <http://bit.ly/1dYuRbS>). Altamente recomendados.

As seqüências binárias ganharão novos construtores e métodos em Python 3.5, com uma das assinaturas atuais de construtor se tornando obsoleta [veja a PEP 467 – *Minor API improvements for binary sequences* (Pequenas melhorias na API para seqüências binárias, <https://www.python.org/dev/peps/pep-0467/>)]. Python 3.5 também deverá ver a implementação da PEP 461 – *Adding % formatting to bytes and bytearray* (Adicionando a formatação com % em bytes e em bytearray, <https://www.python.org/dev/peps/pep-0461/>).

Uma lista de codificações aceitas por Python está disponível em *Standard Encodings* (Codificações-padrões, <https://docs.python.org/3/library/codecs.html#standard-encodings>) na documentação do módulo `codecs`. Se precisar obter essa lista por meio de programação, veja como isso é feito no script `/Tools/unicodelistcodecs.py` (<http://bit.ly/1lqKrQD>) que acompanha o código-fonte de CPython.

Os textos “Changing the Python Default Encoding Considered Harmful” (Alterando a codificação default de Python considerada prejudicial, <http://bit.ly/1lqKu5I>) de Martijn Faassen e “`sys.setdefaultencoding` Is Evil” (`sys.setdefaultencoding` é do mal, <http://blog.ziade.org/2008/01/08/syssetdefaultencoding-is-evil/>) de Tarek Ziadé explicam por que a codificação default obtida de `sys.getdefaultencoding()` jamais deve ser alterada, mesmo que você descubra como fazê-lo.

Os livros *Unicode Explained* (<http://shop.oreilly.com/product/9780596101213.do>) de Jukka K. Korpela (O’Reilly) e *Unicode Demystified* (<http://bit.ly/1dYveDI>) de Richard Gillam (Addison-Wesley) não são específicos para Python, mas foram muito úteis quando estudei os conceitos de Unicode. *Programming with Unicode* (<http://unicodebook.readthedocs.org/index.html>)

¹³ N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

de Victor Stinner é um livro gratuito, publicado por conta própria (Creative Commons BY-SA), que discute Unicode em geral, assim como aborda ferramentas e APIs no contexto dos principais sistemas operacionais e de algumas linguagens de programação, incluindo Python.

As páginas *Case Folding: An Introduction* (Case folding: uma introdução, http://www.w3.org/International/wiki/Case_folding) e *Character Model for the World Wide Web: String Matching and Searching* (Modelo de caracteres para a World Wide Web: correspondência e pesquisa de strings, <http://www.w3.org/TR/charmod-norm/>) do W3C discutem conceitos de normalização, com o primeiro contendo uma introdução leve e o último, uma versão preliminar em andamento, escrita em um estilo seco – o mesmo tom de *Unicode Standard Annex #15 – Unicode Normalization Forms* (Anexo 15 do padrão Unicode – Formas de normalização de Unicode, <http://unicode.org/reports/tr15/>). O *Frequently Asked Questions / Normalization* (Perguntas frequentes/Normalização, <http://www.unicode.org/faq/normalization.html>) de Unicode.org (<http://www.unicode.org/>) é mais legível, assim como o NFC FAQ (<http://www.macchiato.com/unicode/nfc-faq>) de Mark Davis – autor de diversos algoritmos para Unicode e presidente do Unicode Consortium quando esta obra foi escrita.

Ponto de vista

O que é “texto puro” (plain text)?

Para qualquer pessoa que lide com texto que não esteja em inglês em seu cotidiano, “texto puro” (*plain text*) não implica em “ASCII puro”. O *Unicode Glossary* (Glossário do Unicode, http://www.unicode.org/glossary/#plain_text) define *plain text* da seguinte maneira:

Texto codificado pelo computador que consiste somente de uma sequência de pontos de código de um dado padrão, sem outras formatações ou informações estruturais.

Essa definição começa muito bem, mas não concordo com a parte que vem depois da vírgula. HTML é um ótimo exemplo de um formato texto puro que tem informações de formatação e informações estruturais. Porém ele continua sendo um texto puro porque todo byte em um arquivo desse tipo está presente para representar um caractere de texto, normalmente usando UTF-8. Não há nenhum byte com significado que não seja textual, como você pode encontrar em um documento *.png* ou *.xls*, em que a maioria dos bytes representa valores binários compactos, como valores RGB e números de ponto flutuante. Em um texto puro, os números são representados como sequências de caracteres que são dígitos.

Estou escrevendo este livro em um formato texto puro chamado – ironicamente – de AsciiDoc (<http://www.methods.co.nz/asciidoc/>), que faz parte do conjunto de ferramentas do Atlas (<https://atlas.oreilly.com/>), uma excelente plataforma para publicação de livros da O'Reilly. Os arquivos-fontes de AsciiDoc são textos puros, mas são UTF-8, e não ASCII. Se não fosse por isso, escrever este capítulo teria sido realmente penoso. Apesar do nome, o AsciiDoc é realmente muito bom.

O mundo do Unicode está constantemente se expandindo e, nas fronteiras, o suporte das ferramentas nem sempre está presente. É por isso que precisei usar imagens para as figuras 4.1, 4.3 e 4.4; nem todos os caracteres que eu queria mostrar estavam disponíveis nas fontes usadas para gerar o livro. Por outro lado, os terminais de Ubuntu 14.04 e de OSX 10.9 exibem esses caracteres perfeitamente – incluindo os caracteres japoneses da palavra “mojibake”: 文字化け.

Enigmas do Unicode

Qualificadores imprecisos como “frequentemente”, “a maioria” e “geralmente” parecem surgir sempre que escrevo sobre normalização de Unicode. Sinto não ter conselhos mais definitivos, mas há tantas exceções às regras em Unicode que é difícil ser absolutamente categórico.

Por exemplo, μ (sinal de micro) é considerado um “caractere de compatibilidade”, porém os símbolos Ω (ohm) e Å (Ångström) não são. A diferença tem consequências práticas: a normalização NFC – recomendada para comparação de texto – substitui Ω (ohm) por Ω (letra grega ômega maiúscula) e Å (Ångström) por Å (A maiúsculo com um anel em cima). Porém, como “caractere de compatibilidade”, μ (sinal de micro) não é substituído pelo visualmente idêntico μ (letra grega mu minúscula), exceto quando as normalizações NFKC e NFKD mais robustas são aplicadas, e essas transformações causam perdas.

Entendo que μ (sinal de micro) está em Unicode porque aparece na codificação latin1, e substituí-lo pela letra grega mu causaria problemas na conversão de um para outro. Afinal de contas, é por isso que o sinal de micro é um “caractere de compatibilidade”. No entanto, se os símbolos para ohm e Ångström não estão no Unicode por questões de compatibilidade, então por que eles existem? Já há códigos Unicode para GREEK CAPITAL LETTER OMEGA e LATIN CAPITAL LETTER A WITH RING ABOVE, que parecem iguais e os substituem na normalização NFC. Vá entender.

Minha interpretação após muitas horas estudando Unicode é esta: ele é extremamente complexo e cheio de casos especiais que refletem a maravilhosa variedade de línguas humanas e a politicagem das normas técnicas.

Como str é representado em RAM?

A documentação oficial de Python evita a questão de como os códigos Unicode de uma `str` são armazenados na memória. Afinal de contas, isso é um detalhe de implementação. Teoricamente, não importa: qualquer que seja a representação interna, todo `str` deve ser codificado para bytes na saída.

Em memória, Python 3 armazena cada `str` como uma sequência de códigos Unicode usando um número fixo de bytes por código para permitir um acesso direto eficiente a qualquer caractere ou fatia.

Antes de Python 3.3, CPython podia ser compilado para usar 16 ou 32 bits por código Unicode em RAM; a primeira opção era uma “narrow build”, enquanto a última era uma “wide build”. Para saber qual delas você tem, verifique o valor de `sys.maxunicode`: 65535 implica uma “narrow build” que não é capaz de tratar códigos Unicode acima de U+FFFF de modo transparente. Uma “wide build” não tem essa limitação, porém consome bastante memória: 4 bytes por caractere, mesmo que a grande maioria dos códigos Unicode para ideogramas chineses caiba em 2 bytes. Nenhuma opção era ótima, portanto você tinha que escolher de acordo com suas necessidades.

A partir de Python 3.3, ao criar cada novo objeto `str`, o interpretador verifica os caracteres contidos nele e seleciona o layout de memória mais econômico, adequado a esse `str` em particular: se houver caracteres apenas no intervalo `latin1`, esse `str` usará somente um byte por código Unicode. Caso contrário, 2 ou 4 bytes por código Unicode poderão ser usados, dependendo do `str`. Essa é uma simplificação; para obter os detalhes completos, dê uma olhada na *PEP 393 – Flexible String Representation* (Representação flexível de strings, <https://www.python.org/dev/peps/pep-0393/>).

A representação flexível de strings é semelhante ao modo como o tipo `int` funciona em Python 3; se o inteiro couber em uma palavra do computador, ele será armazenado em uma palavra do computador. Caso contrário, o interpretador alternará para uma representação de tamanho variado, como usada no tipo `long` em Python 2. É bom ver boas ideias se espalhando.

PARTE III

Funções como objetos

CAPÍTULO 5

Funções de primeira classe

Jamais considerei que Python tenha sido altamente influenciado por linguagens funcionais, independentemente do que as pessoas dizem ou pensam. Eu tinha muito mais familiaridade com linguagens imperativas como C e Algol 68 e, apesar de ter tornado as funções objetos de primeira classe, não imaginei Python como uma linguagem de programação funcional.¹

— Guido van Rossum

Python BDFL

Funções em Python são objetos de primeira classe. Os teóricos das linguagens de programação definem um “objeto de primeira classe” como uma entidade que pode ser:

- criada em tempo de execução;
- atribuída a uma variável ou a um elemento em uma estrutura de dados;
- passada como argumento a uma função;
- devolvida como resultado de uma função.

Inteiros, strings e dicionários são outros exemplos de objetos de primeira classe em Python – nenhuma grande novidade até aqui. Porém, se você chegou a Python a partir de uma linguagem em que as funções não são cidadãs de primeira classe, este capítulo e o restante da Parte III do livro enfoca as implicações e aplicações práticas do tratamento de funções como objetos.



O termo “funções de primeira classe” é amplamente usado como forma abreviada para “funções como objetos de primeira classe”. Não é perfeito porque parece sugerir uma “elite” entre as funções. Em Python, todas as funções são de primeira classe.

¹ “Origins of Python’s *Functional* Features” (Origem dos recursos funcionais de Python, <http://bit.ly/1FHfhlo>) do blog *The History of Python* de Guido.

Tratando uma função como um objeto

A sessão de console no exemplo 5.1 mostra que as funções Python são objetos. Nesse caso, criamos, chamamos a função, lemos o seu atributo `__doc__` e verificamos que o objeto função propriamente dito é uma instância da classe `function`.

Exemplo 5.1 – Cria e testa uma função; em seguida, lê seu `__doc__` e verifica seu tipo

```
>>> def factorial(n): ❶
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
...
>>> factorial(42)
14050061177528798985431426062445115699363840000000000
>>> factorial.__doc__ ❷
'returns n!'
>>> type(factorial) ❸
<class 'function'>
```

- ❶ Essa é uma sessão de console, portanto estamos criando uma função em “tempo de execução”.
- ❷ `__doc__` é um dos diversos atributos de objetos-função.
- ❸ `factorial` é uma instância da classe `function`.

O atributo `__doc__` é usado para gerar o texto de ajuda de um objeto. No console interativo de Python, o comando `help(factorial)` exibirá uma tela como a da figura 5.1.

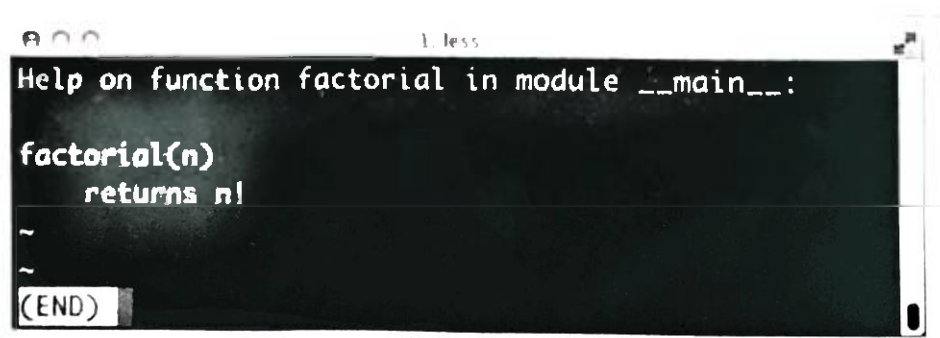


Figura 5.1 – Tela de ajuda da função `factorial`; o texto é do atributo `__doc__` do objeto-função.

O exemplo 5.2 mostra a natureza de “primeira classe” de um objeto-função. Podemos atribuí-lo a uma variável `fact` e chamá-lo por esse nome. Também podemos passar `factorial` como argumento para `map`. A função `map` devolve um iterável em que cada item é o resultado da aplicação do primeiro argumento (uma função) a elementos sucessivos do segundo argumento (um iterável), que é `range(10)` nesse exemplo.

Exemplo 5.2 – Usar uma função com um nome diferente e passar a função como argumento.

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Ter funções de primeira classe permite fazer programação em estilo funcional. Uma das marcas registradas da programação funcional é o uso de funções de ordem superior (higher-order functions), que é o nosso próximo assunto.

Funções de ordem superior

Uma função que aceita uma função como argumento ou que devolve uma função como resultado é uma *função de ordem superior* (higher-order function). Um exemplo é `map`, mostrado no exemplo 5.2. Outro exemplo é a função embutida `sorted`: um argumento opcional `key` permite especificar uma função a ser aplicada a cada item para ordenação, conforme vimos na seção “`list.sort` e a função embutida `sorted`” na página 69.

Por exemplo, para ordenar uma lista de palavras de acordo com o tamanho, basta passar a função `len` em `key`, como mostra o exemplo 5.3.

Exemplo 5.3 – Ordenando uma lista de palavras de acordo com o tamanho

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

Qualquer função de um argumento pode ser usada como chave. Por exemplo, para criar um dicionário de rimas, talvez seja conveniente ordenar as palavras grafadas de trás para frente. No exemplo 5.4, observe que as palavras da lista não são, de forma alguma, alteradas; apenas a sua grafia inversa é usada como critério de ordenação para que as “berries” apareçam juntas.

Exemplo 5.4 – Ordenando uma lista de palavras de acordo com a sua grafia inversa

```
>>> def reverse(word):
...     return word[::-1]
```

```
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

No paradigma da programação funcional, algumas das funções de ordem superior mais conhecidas são `map`, `filter`, `reduce` e `apply`. A função `apply` tornou-se obsoleta em Python 2.3 e foi removida em Python 3 porque não era mais necessária. Se precisar chamar uma função com um conjunto dinâmico de argumentos, basta escrever `fn(*args, **kwargs)` em vez de `apply(fn, args, kwargs)`.

As funções de ordem superior `map`, `filter` e `reduce` continuam presentes, porém alternativas melhores estão disponíveis para a maioria dos casos de uso, como veremos na próxima seção.

Substitutos modernos para `map`, `filter` e `reduce`

As linguagens funcionais normalmente oferecem as funções de ordem superior `map`, `filter` e `reduce` (às vezes, com nomes diferentes). As funções `map` e `filter` continuam sendo funções embutidas em Python 3, mas, desde a introdução das `list comprehensions` e das expressões geradoras, elas não são mais tão importantes. Uma `listcomp` ou uma `genexp` fazem o trabalho conjunto de `map` e `filter`, porém são mais legíveis. Considere o exemplo 5.5.

Exemplo 5.5 – Lista de fatoriais gerada com `map` e `filter`, em comparação com alternativas escritas com `list comprehensions`

```
>>> list(map(fact, range(6))) ❶
[1, 1, 2, 6, 24, 120]
>>> [fact(n) for n in range(6)] ❷
[1, 1, 2, 6, 24, 120]
>>> list(map(factorial, filter(lambda n: n % 2, range(6)))) ❸
[1, 6, 120]
>>> [factorial(n) for n in range(6) if n % 2] ❹
[1, 6, 120]
>>>
```

- ❶ Cria uma lista de fatoriais de 0! a 5!.
- ❷ Mesma operação com uma `list comprehension`.
- ❸ Lista de fatoriais de números ímpares até 5! usando tanto `map` quanto `filter`.
- ❹ `List comprehension` faz a mesma tarefa substituindo `map` e `filter` e tornando `lambda` desnecessário.

Em Python 3, `map` e `filter` devolvem geradores – uma forma de iterador –, portanto seu substituto direto agora é uma expressão geradora (em Python 2, essas funções devolviam listas e, sendo assim, a alternativa mais próxima é uma `listcomp`).

A função `reduce` deixou de ser uma função embutida em Python 2 e passou para o módulo `functools` em Python 3. Seu caso de uso mais comum, a soma, é mais bem tratado pela função embutida `sum`, disponível desde Python 2.3, lançado em 2003. Foi uma grande vitória em termos de legibilidade e de desempenho (veja o exemplo 5.6).

Exemplo 5.6 – Soma de inteiros até 99 realizada com `reduce` e `sum`

```
>>> from functools import reduce ❶
>>> from operator import add ❷
>>> reduce(add, range(100)) ❸
4950
>>> sum(range(100)) ❹
4950
>>>
```

- ❶ A partir de Python 3.0, `reduce` deixou de ser uma função embutida.
- ❷ Importa `add` para evitar a criação de uma função somente para somar dois números.
- ❸ Soma inteiros até 99.
- ❹ Mesma tarefa usando `sum`; importação ou função para adição não são necessárias.

A ideia comum de `sum` e `reduce` é aplicar alguma operação a itens sucessivos em uma sequência, acumulando resultados anteriores e, desse modo, reduzindo uma sequência de valores a um único valor.

Outras funções embutidas para redução são `all` e `any`:

`all(iterable)`

Devolve `True` se todos os elementos de `iterable` forem verdadeiros; `all([])` devolve `True`.

`any(iterable)`

Devolve `True` se algum elemento de `iterable` for verdadeiro; `any([])` devolve `False`.

Darei uma explicação mais completa de `reduce` na seção “Vector Tomada #4: hashing e um `==` mais rápido” na página 332, em que um exemplo em andamento oferecerá um contexto significativo para o uso dessa função. As funções de redução serão resumidas posteriormente no livro, quando os iteráveis estiverem em foco na seção “Funções de redução de iteráveis” na página 486.

Para usar uma função de ordem superior, às vezes, é conveniente criar uma pequena função usada apenas uma vez. É por isso que as funções anônimas existem. Elas serão discutidas a seguir.

Funções anônimas

A palavra reservada `lambda` cria uma função anônima em uma expressão Python.

Entretanto a sintaxe simples de Python limita o corpo de funções `lambda` a serem expressões puras. Em outras palavras, o corpo de uma `lambda` não pode fazer atribuições nem usar nenhum outro comando Python como `while`, `try` etc.

O melhor uso das funções anônimas está no contexto de uma lista de argumentos. No exemplo 5.7, temos o exemplo do índice de rimas do exemplo 5.4 reescrito com `lambda`, sem definir uma função `reverse`.

Exemplo 5.7 – Ordenando uma lista de palavras de acordo com as grafias inversas usando `lambda`

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

Fora do contexto limitado de argumentos para funções de ordem superior, as funções anônimas raramente têm utilidade em Python. As restrições sintáticas tendem a deixar lambdas não triviais ilegíveis ou impraticáveis.

Método Lundh para refatorar lambdas

Se você encontrar um trecho de código difícil de entender por causa de uma `lambda`, Fredrik Lundh sugere o procedimento de refatoração a seguir:

1. Escreva um comentário explicando que cargas d'água esse `lambda` faz.
2. Estude o comentário por um instante e pense em um nome que capture a essência do comentário.
3. Converta o `lambda` em um comando `def` usando esse nome.
4. Remova o comentário.

Esses passos foram copiados de *Functional Programming HOWTO* (HOWTO para programação funcional, <http://docs.python.org/3/howto/functional.html>) – uma leitura obrigatória.

A sintaxe de `lambda` é somente um açúcar sintático: uma expressão `lambda` cria um objeto função assim como o comando `def`. Esse é apenas um dos diversos tipos de objetos invocáveis (callable) em Python. A seção a seguir analisa todos eles.

As sete variações de objetos invocáveis

O operador de invocação (isto é, `()`, o *call operator*) pode ser aplicado a outros objetos além de funções definidas pelo usuário. Para determinar se um objeto é invocável (callable), use a função embutida `callable()`. A documentação de Python Data Model (Modelo de dados de Python) lista sete tipos de invocáveis:

Funções definidas pelo usuário

Criadas com comandos `def` ou expressões `lambda`.

Funções embutidas

Uma função implementada em C (no caso do CPython), como `len` ou `time.strftime`.

Métodos embutidos

Métodos implementados em C, como `dict.get`.

Métodos

Funções definidas no corpo de uma classe do usuário.

Classes

Quando chamada, uma classe executa o seu método `__new__` para criar uma instância e, em seguida, `__init__` para inicializá-la e, por fim, a instância é devolvida a quem chamou. Como não há um operador `new` em Python, chamar uma classe é como chamar uma função. (Geralmente, chamar uma classe cria uma instância dessa mesma classe, mas outros comportamentos são possíveis se `__new__` for sobrescrito. Veremos um exemplo desse caso na seção “Criação flexível de objetos com `__new__`” na página 652.)

Instâncias de classe

Se uma classe define um método `__call__`, suas instâncias poderão ser chamadas como funções. Veja a seção “Tipos invocáveis definidos pelo usuário” na página 182.

Funções geradoras

São funções ou métodos que utilizam a palavra reservada `yield`. Quando chamadas, as funções geradoras devolvem um objeto gerador.

As funções geradoras são diferentes de outros invocáveis em vários aspectos. O capítulo 14 será dedicado a elas. Essas funções também podem ser usadas como corrotinas, que serão discutidas no capítulo 16.



Dada a variedade de tipos invocáveis existentes em Python, a maneira mais segura de determinar se um objeto é invocável é usando a função embutida `callable()`:

```
>>> abs, str, 13
(<built-in function abs>, <class 'str'>, 13)
>>> [callable(obj) for obj in (abs, str, 13)]
[True, True, False]
```

Passaremos agora para a criação de instâncias de classe que funcionam como objetos invocáveis.

Tipos invocáveis definidos pelo usuário

Não só as funções Python são objetos reais como também podemos fazer objetos Python quaisquer se comportarem como funções. Basta implementar um método de instância `__call__`.

O exemplo 5.8 implementa uma classe `BingoCage`. Uma instância é criada a partir de qualquer iterável e armazena uma `list` interna de itens em ordem aleatória. Chamar a instância faz um item ser extraído.

Exemplo 5.8 – `bingocall.py`: um `BingoCage` realiza uma única tarefa – seleciona itens de uma lista embaralhada

```
import random

class BingoCage:
    def __init__(self, items):
        self._items = list(items) ❶
        random.shuffle(self._items) ❷
    def pick(self): ❸
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage') ❹
    def __call__(self): ❺
        return self.pick()
```

- ❶ `__init__` aceita qualquer iterável; criar uma cópia local evita efeitos colaterais inesperados em qualquer `list` passada como argumento.

- ❷ É certo que `shuffle` funcionará porque `self._items` é uma `list`.
- ❸ O método principal.
- ❹ Levanta uma exceção com uma mensagem personalizada se `self._items` estiver vazio.
- ❺ Atalho para `bingo.pick()`: `bingo()`.

Eis uma demo simples do exemplo 5.8. Observe como uma instância `bingo` pode ser chamada como uma função, e a função embutida `callable(...)` a reconhece como um objeto invocável:

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

Uma classe que implementa `__call__` é uma maneira fácil de criar objetos do tipo função ou similar com algum estado interno que deva ser mantido entre chamadas, por exemplo, os itens restantes em `BingoCage`. Um exemplo é um decorador (decorator). Os decoradores devem ser funções, mas, às vezes, será conveniente poder “lembrar-se” de alguma informação entre as chamadas do decorador [por exemplo, para memoização – caching de resultados de processamentos intensivos para uso posterior].

Uma abordagem totalmente diferente para criar funções com estado interno é usar closures. As closures, assim como os decoradores, são o assunto do capítulo 7.

Vamos agora passar para outro aspecto do tratamento de funções como objetos: introspecção (introspection) em tempo de execução.

Introspecção de função

Os objetos-função têm diversos atributos além de `__doc__`. Veja o que a função `dir` revela sobre o nosso `factorial`:

```
>>> dir(factorial)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
 '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__get__', '__getattr__', '__globals__',
 '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
 '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
```

```
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
```

A maioria desses atributos é comum aos objetos Python em geral. Nesta seção, discutiremos aqueles que são especialmente relevantes ao tratamento de funções como objetos, começando por `__dict__`.

Assim como as instâncias de uma classe comum definida pelo usuário, uma função usa o atributo `__dict__` para armazenar os atributos de usuário atribuídos a ela. Isso é útil como uma forma primitiva de anotação. A atribuição de atributos quaisquer a funções não é uma prática muito comum em geral, mas Django é um framework que faz uso disso. Veja, por exemplo, os atributos `short_description`, `boolean` e `allow_tags` descritos na documentação em *The Django admin site* (o site do administrador de Django, <https://docs.djangoproject.com/en/1.5/ref/contrib/admin>). Na documentação de Django, o exemplo a seguir mostra a associação de um `short_description` a um método para determinar a descrição que aparecerá em listagens de registros na interface administrativa de Django quando esse método for usado:

```
def upper_case_name(obj):
    return ("%s %s" % (obj.first_name, obj.last_name)).upper()
upper_case_name.short_description = 'Customer name'
```

Vamos agora focar nos atributos específicos de funções, que não são encontrados em um objeto Python genérico definido pelo usuário. Calcular a diferença entre dois conjuntos rapidamente nos dá uma lista dos atributos específicos de funções (veja o exemplo 5.9).

Exemplo 5.9 – Listando os atributos de funções que não existem em instâncias simples

```
>>> class C: pass # ❶
>>> obj = C() # ❷
>>> def func(): pass # ❸
>>> sorted(set(dir(func)) - set(dir(obj))) # ❹
['_annotations_', '__call__', '__closure__', '__code__', '__defaults__',
 '__get__', '__globals__', '__kwdefaults__', '__name__', '__qualname__']
>>>
```

- ❶ Cria uma classe vazia definida pelo usuário.
- ❷ Cria uma instância dessa classe.
- ❸ Cria uma função vazia.
- ❹ Usando a diferença entre conjuntos, gera uma lista ordenada de atributos existentes em uma função, mas não em uma instância de uma classe vazia.

A tabela 5.1 mostra um resumo dos atributos listados pelo exemplo 5.9.

Tabela 5.1 – Atributos de funções definidos pelo usuário

Nome	Tipo	Descrição
<code>__annotations__</code>	dict	Anotações de parâmetros e de valor de retorno
<code>__call__</code>	method-wrapper	Implementação do operador <code>()</code> , também conhecido como protocolo de objetos invocáveis
<code>__closure__</code>	tuple	A closure da função, ou seja, associações para variáveis livres (frequentemente é <code>None</code>)
<code>__code__</code>	code	Metadados e corpo da função compilados em <code>bytecode</code>
<code>__defaults__</code>	tuple	Valores default para os parâmetros formais
<code>__get__</code>	method-wrapper	Implementação do protocolo de descritor somente para leitura (veja o capítulo 20)
<code>__globals__</code>	dict	Variáveis globais do módulo em que a função é definida
<code>__kwdefaults__</code>	dict	Valores default para os parâmetros formais exclusivamente nomeados
<code>__name__</code>	str	O nome da função
<code>__qualname__</code>	str	O nome qualificado da função, por exemplo, <code>Random.choice</code> (consulte a PEP-3155 em https://www.python.org/dev/peps/pep-3155/)

Discutiremos as funções `__defaults__`, `__code__` e `__annotations__` usadas por IDEs e frameworks para extrair informações sobre assinaturas de função em seções mais adiante. Contudo, para uma apreciação completa desses atributos, faremos um desvio para explorar a sintaxe poderosa oferecida por Python para declarar parâmetros de função e passar argumentos a eles.

De parâmetros posicionais a parâmetros exclusivamente nomeados

Um dos melhores recursos das funções Python é o mecanismo extremamente flexível de tratamento de parâmetros, aperfeiçoado com argumentos exclusivamente nomeados em Python 3. Relacionados intimamente a esse assunto estão os usos de `*` e de `**` para “explodir” iteráveis e mapeamentos em argumentos separados quando chamamos uma função. Para ver esses recursos em ação, observe o código do exemplo 5.10 e os testes que mostram seu uso no exemplo 5.11.

Exemplo 5.10 – A função `tag` gera HTML; um argumento `cls` exclusivamente nomeado é usado para passar atributos “class” como uma solução alternativa, pois `class` é uma palavra reservada em Python

```
def tag(name, *content, cls=None, **attrs):
    """Gera uma ou mais tags HTML"""
    if cls is not None:
        attrs['class'] = cls
```

```

if attrs:
    attr_str = ''.join(' %s="%s"' % (attr, value)
                        for attr, value
                        in sorted(attrs.items()))

else:
    attr_str = ''

if content:
    return '\n'.join('<%s%s>%s</%s>' %
                    (name, attr_str, c, name) for c in content)

else:
    return '<%s%s />' % (name, attr_str)

```

A função `tag` pode ser chamada de várias maneiras, como mostra o exemplo 5.11.

Exemplo 5.11 – Algumas das várias maneiras de chamar a função `tag` do exemplo 5.10

```

>>> tag('br') ❶
'<br />'

>>> tag('p', 'hello') ❷
'<p>hello</p>'

>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>

>>> tag('p', 'hello', id=33) ❸
'<p id="33">hello</p>'

>>> print(tag('p', 'hello', 'world', cls='sidebar')) ❹
<p class="sidebar">hello</p>
<p class="sidebar">world</p>

>>> tag(content='testing', name="img") ❺
'<img content="testing" />'

>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...          'src': 'sunset.jpg', 'cls': 'framed'}
>>> tag(**my_tag) ❻
''

```

- ❶ Um único argumento posicional produz uma tag vazia com esse nome.
- ❷ Qualquer quantidade de argumentos após o primeiro é capturada por `*content` como uma tuple.
- ❸ Argumentos nomeados não explicitamente nomeados na assinatura de `tag` são capturados por `**attrs` como um dict.
- ❹ O parâmetro `cls` pode ser passado somente como um argumento nomeado.

- ❸ Mesmo o primeiro argumento posicional pode ser passado como um argumento nomeado quando `tag` é chamada.
- ❹ Prefixar o dict `my_tag` com `**` faz todos os seus itens serem passados como argumentos separados, que são então associados aos parâmetros nomeados, com o restante capturado por `**attrs`.

Argumentos exclusivamente nomeados são um recurso novo em Python 3. No exemplo 5.10, o parâmetro `cls` pode ser especificado somente como um argumento nomeado – ele jamais capturará argumentos posicionais não nomeados. Para especificar os argumentos exclusivamente nomeados ao definir uma função, nomeie-os após o argumento prefixado com `*`. Se não quiser aceitar argumentos posicionais variáveis, mas ainda quiser ter argumentos exclusivamente nomeados, coloque um `*` sozinho na assinatura, desta maneira:

```
>>> def f(a, *, b):  
...     return a, b  
...  
>>> f(1, b=2)  
(1, 2)
```

Observe que os argumentos exclusivamente nomeados não precisam ter um valor default: eles podem ser obrigatórios, como `b` no exemplo anterior.

Vamos agora passar para a introspecção de parâmetros de função, começando com um exemplo prático de um framework web e prosseguindo com as técnicas de introspecção.

Obtendo informações sobre parâmetros

Uma aplicação interessante da introspecção de função pode ser encontrada no microframework HTTP Bobo. Para vê-la em ação, considere uma variação da aplicação “Hello world” do tutorial do Bobo no exemplo 5.12.

Exemplo 5.12 – Bobo sabe que `hello` exige um argumento `person` e o obtém da requisição HTTP

```
import bobo  
  
@bobo.query('/')  
def hello(person):  
    return 'Hello %s!' % person
```

O decorador `bobo.query` integra uma função simples como `hello` ao mecanismo de tratamento de requisições do framework. Discutiremos os decoradores no capítulo 7 – essa não é a questão principal nesse exemplo. A questão é que Bobo faz uma introspecção

na função `hello` e descobre que precisa de um parâmetro chamado `person` para funcionar; ele obterá um parâmetro com esse nome a partir da requisição e o passará para `hello`, de modo que o programador não precisará ter nenhum contato com o objeto que representa a requisição.

Se você instalar o Bobo e apontar seu servidor de desenvolvimento para o script do exemplo 5.12 (por exemplo, `bobó -f hello.py`), um acesso ao URL `http://localhost:8080/` resultará na mensagem “Missing form variable person” (Variável de formulário `person` ausente) com um código HTTP 403. Isso acontece porque o framework Bobo entende que o argumento `person` é necessário para chamar `hello`, mas esse nome não foi encontrado na requisição. O exemplo 5.13 é uma sessão de shell que usa `curl` para mostrar esse comportamento.

Exemplo 5.13 – Bobo gera uma resposta 403 forbidden se houver argumentos de função ausentes na requisição; `curl -i` é usado para fazer dump dos cabeçalhos na saída-padrão

```
$ curl -i http://localhost:8080/
HTTP/1.0 403 Forbidden
Date: Thu, 21 Aug 2014 21:39:44 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 103

<html>
<head><title>Missing parameter</title></head>
<body>Missing form variable person</body>
</html>
```

Entretanto, se você usar `http://localhost:8080/?person=Jim`, a resposta será a string ‘Hello Jim!’. Veja o exemplo 5.14.

Exemplo 5.14 – Passar o parâmetro `person` é necessário para obter uma resposta OK

```
$ curl -i http://localhost:8080/?person=Jim
HTTP/1.0 200 OK
Date: Thu, 21 Aug 2014 21:42:32 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 10

Hello Jim!
```

Como o framework Bobo sabe quais nomes de parâmetro são necessários à função e se esses parâmetros têm valores default ou não?

Em um objeto função, o atributo `__defaults__` armazena uma tupla com valores default para argumentos posicionais e nomeados. Os defaults para argumentos exclusivamente nomeados aparecem em `__kwdefaults__`. Os nomes dos argumentos, porém, se encontram no atributo `__code__`, que é uma referência a um objeto `code` com vários atributos próprios.

Para mostrar o uso desses atributos, inspecionaremos a função `clip` em um módulo `clip.py`, listado no exemplo 5.15.

Exemplo 5.15 – Função para reduzir uma string ao fazer um corte em um espaço próximo ao tamanho desejado

```
def clip(text, max_len=80):
    """Retorna o texto cortado no último espaço antes ou depois de max_len
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None: # nenhum espaço foi encontrado
        end = len(text)
    return text[:end].rstrip()
```

O exemplo 5.16 mostra os valores de `__defaults__`, `__code__.co_varnames` e `__code__.co_argcount` para a função `clip` exibida no exemplo 5.15.

Exemplo 5.16 – Extraindo informações sobre os argumentos da função

```
>>> from clip import clip
>>> clip.__defaults__
(80,)
>>> clip.__code__ # doctest: +ELLIPSIS
<code object clip at 0x...>
>>> clip.__code__.co_varnames
('text', 'max_len', 'end', 'space_before', 'space_after')
>>> clip.__code__.co_argcount
2
```

Como podemos ver, essa não é a organização mais conveniente para as informações. Os nomes dos argumento aparecem em `__code__.co_varnames`, mas os nomes das variáveis locais criadas no corpo da função também estão incluídos. Desse modo, os nomes dos argumentos são as primeiras N strings, em que N é dado por `__code__.co_argcount`, que – a propósito – não inclui nenhum argumento variável prefixado com `*` ou `**`. Os valores default são identificados somente pela sua posição na tupla `__defaults__`; sendo assim, para associar cada valor com o respectivo argumento, você precisa percorrer os itens do último para o primeiro. No exemplo, temos dois argumentos, `text` e `max_len`, e um default `80`, portanto ele deve pertencer ao último argumento, `max_len`. Isso não é nada prático.

Felizmente, há uma opção melhor: o módulo `inspect`.

Dê uma olhada no exemplo 5.17.

Exemplo 5.17 – Extraíndo a assinatura da função

```
>>> from clip import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig # doctest: +ELLIPSIS
<inspect.Signature object at 0x...>
>>> str(sig)
'(text, max_len=80)'
>>> for name, param in sig.parameters.items():
...     print(param.kind, ':', name, '=', param.default)
...
POSITIONAL_OR_KEYWORD : text = <class 'inspect._empty'>
POSITIONAL_OR_KEYWORD : max_len = 80
```

Isso é muito melhor; `inspect.signature` devolve um objeto `inspect.Signature` com um atributo `parameters` que permite ler um mapeamento ordenado de nomes para objetos `inspect.Parameter`. Toda instância de `Parameter` tem atributos como `name`, `default` e `kind`. O valor especial `inspect._empty` indica parâmetros sem default, o que faz sentido, considerando que `None` é um valor default válido – e popular.

O atributo `kind` armazena um de cinco valores possíveis da classe `_ParameterKind`:

POSITIONAL_OR_KEYWORD

Um parâmetro que pode ser passado como um argumento posicional ou nomeado (a maioria dos parâmetros de funções Python é desse tipo).

VAR_POSITIONAL

Uma tuple de parâmetros posicionais.

VAR_KEYWORD

Um dict de parâmetros nomeados.

KEYWORD_ONLY

Um parâmetro exclusivamente nomeado (novo em Python 3).

POSITIONAL_ONLY

Um parâmetro exclusivamente posicional; no momento, não é aceito pela sintaxe de declaração de funções Python, mas há exemplos de funções existentes implementadas em C – como `divmod` – que não aceitam parâmetros passados de forma nomeada.

Além de `name`, `default` e `kind`, os objetos `inspect.Parameter` têm um atributo `annotation` que geralmente é `inspect._empty`, mas pode conter metadados da assinatura da função, oferecidos pela nova sintaxe de anotações em Python 3 (as anotações serão discutidas na próxima seção).

Um objeto `inspect.Signature` tem um método `bind` que aceita qualquer quantidade de argumentos e os associa aos parâmetros da assinatura, aplicando as regras usuais para correspondência entre argumentos e parâmetros formais. Esse método pode ser usado por um framework para validar os argumentos antes da chamada da função. O exemplo 5.18 mostra como fazer isso.

Exemplo 5.18 – Associando a assinatura da função `tag` do exemplo 5.10 a um dict com os argumentos

```
>>> import inspect
>>> sig = inspect.signature(tag) ❶
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...          'src': 'sunset.jpg', 'cls': 'framed'}
>>> bound_args = sig.bind(**my_tag) ❷
>>> bound_args
<inspect.BindArguments object at 0x...> ❸
>>> for name, value in bound_args.arguments.items(): ❹
...     print(name, '=', value)
...
name = img
cls = framed
attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}
>>> del my_tag['name'] ❺
>>> bound_args = sig.bind(**my_tag) ❻
Traceback (most recent call last):
...
TypeError: 'name' parameter lacking default value
```

- ❶ Obtém a assinatura da função `tag` do exemplo 5.10.
- ❷ Passa um dict com os argumentos para `.bind()`.
- ❸ Um objeto `inspect.BoundsArguments` é produzido.
- ❹ Faz uma iteração pelos itens em `bound_args.arguments`, que é um `OrderedDict`, para exibir os nomes e os valores dos argumentos.
- ❺ Remove o argumento obrigatório `name` de `my_tag`.
- ❻ Chamar `sig.bind(**my_tag)` gera um `TypeError` que reclama do parâmetro `name` ausente.

Esse exemplo mostra como o modelo de dados de Python, com a ajuda de `inspect`, expõe o mesmo mecanismo usado pelo interpretador para associar argumentos a parâmetros formais em chamadas de função.

Frameworks e ferramentas como IDEs podem usar essas informações para validar códigos. Outro recurso de Python 3, as anotações de função, expandem os possíveis usos disso, como veremos a seguir.

Anotações de função

Python 3 oferece uma sintaxe para associar metadados aos parâmetros de uma declaração de função e ao seu valor de retorno. O exemplo 5.19 é uma versão do exemplo 5.15 com anotações. As únicas diferenças estão na primeira linha.

Exemplo 5.19 – Função `clip` com anotações

```
def clip(text:str, max_len:'int > 0'=80) -> str: ❶
    """Retorna o texto cortado no último espaço antes ou depois de max_len
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None: # nenhum espaço foi encontrado
        end = len(text)
    return text[:end].rstrip()
```

- ❶ A declaração da função com anotações.

Cada argumento na declaração de função pode ter uma expressão de anotação precedida por `:`. Se houver um valor default, a anotação deve ser inserida entre o nome do argumento e o sinal `=`. Para uma anotação no valor de retorno, adicione `->` e outra expressão entre `)` e `:` no final da declaração da função. As expressões podem ser de qualquer tipo. Os tipos mais comuns usados em anotações são classes, por exemplo, `str` ou `int`, ou strings, como `'int > 0'`, como vimos na anotação para `max_len` no exemplo 5.19.

Nenhum processamento é feito com as anotações. Elas são simplesmente armazenadas no atributo `__annotations__` da função, que é um dict:

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': 'int > 0', 'return': <class 'str'>}
```

O item com a chave `'return'` armazena a anotação do valor de retorno marcada com `->` na declaração da função do exemplo 5.19.

A coisa que Python faz com as anotações é armazená-las no atributo `__annotations__` da função. Nada mais é feito: não há verificações, garantias, validação nem qualquer outra ação. Em outras palavras, as anotações não têm nenhum significado para o interpretador Python. Elas são apenas metadados que podem ser usados por ferramentas como IDEs, frameworks e decoradores. Atualmente (enquanto escrevo este livro), não há nenhuma ferramenta que use esses metadados na biblioteca-padrão; há somente `inspect.signature()`, que sabe extrair as anotações, como mostra o exemplo 5.20.

Exemplo 5.20 – Extraindo anotações da assinatura da função

```
>>> from clip_annot import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig.return_annotation
<class 'str'>
>>> for param in sig.parameters.values():
...     note = repr(param.annotation).ljust(13)
...     print(note, ': ', param.name, '=', param.default)
<class 'str'> : text = <class 'inspect._empty'>
'int > 0'      : max_len = 80
```

A função `signature` retorna um objeto `Signature`, que tem um atributo `return_annotation` e um dicionário `parameters` que mapeia nomes de parâmetros a objetos `Parameter`. Cada objeto `Parameter` tem seu próprio atributo `annotation`. É assim que o exemplo 5.20 funciona.

No futuro, frameworks como Bobo poderão tratar anotações para automatizar mais ainda o processamento de requisições. Por exemplo, um argumento com uma anotação `price:float` poderá ser convertido automaticamente de uma string de query para o float esperado pela função; um parse poderá ser feito em uma anotação de string como `quantity:'int > 0'` para realizar conversão e validação de um parâmetro.

O maior impacto das anotações de função provavelmente não será em configurações dinâmicas como no framework Bobo, mas na possibilidade de fornecer informações opcionais de tipo para verificação estáticas de tipagem em ferramentas como IDEs e linters.

Após essa imersão na anatomia das funções, o restante deste capítulo discutirá os pacotes mais úteis da biblioteca-padrão que dão suporte à programação funcional.

Pacotes para programação funcional

Embora Guido deixe claro que Python não tem como objetivo ser uma linguagem de programação funcional, um estilo de codificação funcional pode ser usado até de forma ampla, graças ao suporte oferecido por pacotes como `operator` e `functools`, que discutiremos nas duas próximas seções.

Módulo `operator`

Com frequência, na programação funcional, é conveniente usar um operador aritmético como função. Por exemplo, suponha que você queira multiplicar uma sequência de números para calcular fatoriais sem usar recursão. Para efetuar uma soma, podemos usar `sum`, porém não existe uma função equivalente para a multiplicação. Podemos usar `reduce` – como vimos na seção “Substitutos modernos para `map`, `filter` e `reduce`” na página 178 –, mas isso exige uma função para multiplicar dois itens da sequência. O exemplo 5.21 mostra como resolver esse problema usando `lambda`.

Exemplo 5.21 – Fatorial implementado com `reduce` e uma função anônima

```
from functools import reduce

def fact(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

Para evitar o trabalho de escrever funções anônimas triviais como `lambda a, b: a*b`, o módulo `operator` oferece equivalentes de funções para dezenas de operadores aritméticos. Com ele, podemos reescrever o exemplo 5.21 como o exemplo 5.22.

Exemplo 5.22 – Fatorial implementado com reduce e operator.mul

```
from functools import reduce
from operator import mul

def fact(n):
    return reduce(mul, range(1, n+1))
```

Outro grupo de lambdas com função única que `operator` substitui é o das funções para selecionar itens de sequências ou ler atributos de objetos – `itemgetter` e `attrgetter`, na verdade, criam funções personalizadas para isso.

O exemplo 5.23 mostra um uso comum de `itemgetter`: ordenar uma lista de tuplas de acordo com o valor de um campo. No exemplo, as cidades são exibidas de forma ordenada de acordo com o código do país (campo 1). Essencialmente, `itemgetter(1)` faz o mesmo que `lambda fields: fields[1]`: cria uma função que, dada uma coleção, devolve o item no índice 1.

Exemplo 5.23 – Demo de itemgetter para ordenar uma lista de tuplas (dados do exemplo 2.8)

```
>>> metro_data = [
...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
...     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
...     ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))
```

Se você passar vários argumentos de índice para `itemgetter`, a função criada devolverá tuplas com os valores extraídos:

```
>>> cc_name = itemgetter(1, 0)
>>> for city in metro_data:
...     print(cc_name(city))
...
('JP', 'Tokyo')
```

```

('IN', 'Delhi NCR')
('MX', 'Mexico City')
('US', 'New York-Newark')
('BR', 'Sao Paulo')
>>>

```

Como `itemgetter` usa o operador `[]`, ele suporta não apenas sequências, mas também mapeamentos e qualquer classe que implemente `__getitem__`.

Um irmão de `itemgetter` é `attrgetter`, que cria funções para extrair atributos de objetos pelo nome. Se você passar vários nomes de atributos como argumentos a `attrgetter`, ele também devolverá uma tupla de valores. Além disso, se algum nome de argumento contiver um `.` (ponto), `attrgetter` navegará pelos objetos aninhados para obter o atributo. Esses comportamentos estão mostrados no exemplo 5.24. Essa não é a menor das sessões de console, pois precisamos criar uma estrutura aninhada para fazer a demonstração do tratamento de atributos com ponto por `attrgetter`.

Exemplo 5.24 – Demo de `attrgetter` para processar uma lista previamente definida de `namedtuple` chamada `metro_data` (é a mesma lista do exemplo 5.23)

```

>>> from collections import namedtuple
>>> LatLong = namedtuple('LatLong', 'lat long') # ❶
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord') # ❷
>>> metro_areas = [Metropolis(name, cc, pop, LatLong(lat, long)) # ❸
...                 for name, cc, pop, (lat, long) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLong(lat=35.689722,
long=139.691667))
>>> metro_areas[0].coord.lat # ❹
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat') # ❺
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')): # ❻
...     print(name_lat(city)) # ❼
...
('Sao Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)

```

❶ Usa `namedtuple` para definir `LatLong`.

❷ Também define `Metropolis`.

- ❸ Cria a lista `metro_areas` com instâncias de `Metropolis`; observe o desempacotamento da tupla aninhada para extrair (`lat`, `long`) e usar esses valores para criar o `LatLong` para o atributo `coord` de `Metropolis`.
- ❹ Acessa o elemento `metro_areas[0]` internamente para obter sua latitude.
- ❺ Define um `attrgetter` para obter `name` e o atributo aninhado `coord.lat`.
- ❻ Usa `attrgetter` novamente para ordenar a lista de cidades de acordo com a latitude.
- ❼ Usa o `attrgetter` definido em ❺ para mostrar somente o nome da cidade e a latitude.

Eis uma lista parcial de funções definidas em `operator` (os nomes começados com `_` foram omitidos porque, em sua maior parte, são detalhes de implementação):

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imod', 'imul',
'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift',
'is_', 'is_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le',
'length_hint', 'lshift', 'lt', 'methodcaller', 'mod', 'mul', 'ne',
'neg', 'not_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub',
'truediv', 'truth', 'xor']
```

A maioria dos 52 nomes listados é, por si só, evidente. O grupo de nomes prefixados com `i` e o nome de outro operador – por exemplo, `iadd`, `iand` etc. – corresponde aos operadores de atribuição combinada – como `+=`, `&=` etc. Essa operação altera o primeiro argumento in-place se ele for mutável; se não for, a função atuará como a versão que não tem o prefixo `i` e simplesmente devolverá o resultado da operação.

Entre as funções restantes de `operator`, `methodcaller` é o último que discutiremos. De certo modo, ela é semelhante a `attrgetter` e a `itemgetter` porque cria uma função durante a execução. A função criada chama um método pelo nome no objeto fornecido como argumento, como mostra o exemplo 5.25.

Exemplo 5.25 – Demo de `methodcaller`: o segundo teste mostra a associação com argumentos extras

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
'THE TIME HAS COME'
>>> hiphenate = methodcaller('replace', ' ', '-')
>>> hiphenate(s)
'The-time-has-come'
```

O primeiro teste no exemplo 5.25 está presente somente para mostrar `methodcaller` em funcionamento, mas, se você precisar usar `str.upper` como uma função, basta instalá-la na classe `str` e passar uma string como argumento, da seguinte maneira:

```
>>> str.upper(s)
'THE TIME HAS COME'
```

O segundo teste no exemplo 5.25 mostra que `methodcaller` também pode fazer uma aplicação parcial para congelar alguns argumentos, como faz a função `functools.partial`. Este será o nosso próximo assunto.

Congelando argumentos com `functools.partial`

O módulo `functools` reúne uma porção de funções de ordem superior. A mais conhecida delas provavelmente é `reduce`, discutida na seção “Substitutos modernos para `map`, `filter` e `reduce`” na página 178. Entre as funções restantes em `functools`, as mais úteis são `partial` e a sua variante `partialmethod`.

`functools.partial` é uma função de ordem superior que permite a aplicação parcial de uma função. Dada uma função, uma aplicação parcial gera um novo invocável com alguns dos argumentos da função original fixos. É conveniente para adaptar uma função que aceite um ou mais argumentos em uma API que exija uma callback com menos argumentos. O exemplo 5.26 apresenta uma demonstração trivial.

Exemplo 5.26 – Usando `partial` para utilizar uma função de dois argumentos em que um invocável de um argumento é exigido

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3) ❶
>>> triple(7) ❷
21
>>> list(map(triple, range(1, 10))) ❸
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

❶ Cria uma nova função `triple` a partir de `mul`, associando o primeiro argumento posicional a 3.

❷ Testa.

❸ Usa `triple` com `map`; `mul` não funcionaria com `map` nesse exemplo.

Um exemplo mais útil envolve a função `unicode.normalize`, que vimos na seção “Normalizando Unicode para comparações mais seguras” na página 150. Se você trabalha com texto em vários idiomas, talvez queira aplicar `unicode.normalize('NFC', s)` a qualquer

string `s` antes de fazer comparações ou de armazená-la. Se fizer isso com frequência, será conveniente ter uma função `nfc` que faça essa tarefa, como mostra o exemplo 5.27.

Exemplo 5.27 – Criando uma função conveniente para normalização de Unicode com `partial`

```
>>> import unicodedata, functools
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'café\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

`partial` aceita um invocável como primeiro argumento, seguido de um número arbitrário de argumentos posicionais e nomeados para associar.

O exemplo 5.28 mostra o uso de `partial` com a função `tag` do exemplo 5.10 para congelar um argumento posicional e um argumento nomeado.

Exemplo 5.28 – Demo de `partial` aplicado à função `tag` do exemplo 5.10.

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0> ❶
>>> from functools import partial
>>> picture = partial(tag, 'img', cls='pic-frame') ❷
>>> picture(src='wumpus.jpeg')
'' ❸
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img', cls='pic-frame') ❹
>>> picture.func ❺
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
>>> picture.keywords
{'cls': 'pic-frame'}
```

- ❶ Importa `tag` do exemplo 5.10 e mostra o seu ID.
- ❷ Cria a função `picture` a partir de `tag` fixando o primeiro argumento posicional com `'img'` e o argumento nomeado `cls` com `'pic-frame'`.
- ❸ `picture` funciona conforme esperado.

- ❹ `partial()` devolve um objeto `functools.partial`.²
- ❺ Um objeto `functools.partial` tem atributos que possibilitam ter acesso à função original e aos argumentos fixos.

A função `functools.partialmethod` (nova em Python 3.4) faz o mesmo que `partial`, porém foi projetada para trabalhar com métodos.

Uma função impressionante de `functools` é `lru_cache`, que faz memoização – uma forma de otimização automática que funciona armazenando os resultados de chamadas de função para evitar cálculos repetidos e custosos. Discutiremos esse assunto no capítulo 7, quando explicaremos os decoradores, juntamente com outras funções de ordem superior concebidas para serem usadas como decoradores: `singledispatch` e `wraps`.

Resumo do capítulo

O objetivo deste capítulo foi explorar a natureza de primeira classe das funções em Python. As ideias principais são a possibilidade de atribuir funções a variáveis, passá-las para outras funções, armazená-las em estruturas de dados e acessar atributos de função, permitindo que frameworks e ferramentas atuem sobre essas informações. Funções de ordem superior, que são uma marca registrada da programação funcional, são comuns em Python – mesmo que o uso de `map`, `filter` e `reduce` não sejam mais tão frequentes – graças às `list comprehensions` (e a construções semelhantes como as expressões geradoras) e ao surgimento de funções embutidas de redução como `sum`, `all` e `any`. As funções embutidas `sorted`, `min`, `max` e `functools.partial` são exemplos de funções de ordem superior comumente usadas na linguagem.

Os invocáveis têm sete variantes diferentes em Python, de funções simples criadas com `lambda` a instâncias de classes que implementem `__call__`. Todas elas podem ser detectadas pela função embutida `callable()`. Todo invocável aceita a mesma sintaxe rica para declaração de parâmetros formais, incluindo parâmetros exclusivamente nomeados e anotações – ambos são recursos novos introduzidos em Python 3.

As funções Python e suas anotações têm um conjunto rico de atributos que podem ser lidos com a ajuda do módulo `inspect`; esse módulo inclui o método `Signature`. `bind` para aplicar as regras flexíveis usadas por Python para associar argumentos aos parâmetros declarados.

² O código-fonte (<http://bit.ly/1Vm8cqQ>) de `functools.py` mostra que a classe `functools.partial` é implementada em C e é usada por padrão. Se ela não estiver disponível, há uma implementação de `partial` em Python puro no módulo `functools` a partir de Python 3.4.

Por fim, discutimos algumas funções dos módulos `operator` e `functools.partial`, que facilitam o uso da programação funcional ao minimizar a necessidade de utilizar a sintaxe exigente de `lambda`.

Leituras complementares

Os dois próximos capítulos dão continuidade à nossa exploração de programação com objetos-função. O capítulo 6 mostra como as funções de primeira classe podem simplificar alguns padrões de projeto clássicos orientado a objetos, enquanto o capítulo 7 explora os decoradores de função – um tipo especial de função de ordem superior – e o mecanismo de closure que as faz funcionar.

O capítulo 7 de *Python Cookbook*, 3ª edição (O'Reilly)³ de David Beazley e Brian K. Jones é um complemento excelente para este capítulo, assim como para o capítulo 7 deste livro, e discute basicamente os mesmos conceitos com uma abordagem diferente.

Em *The Python Language Reference* (Guia de referência à linguagem Python), a seção “3.2. The standard type hierarchy” (A hierarquia padrão de tipos, <http://bit.ly/1Vm8dv2>) apresenta os sete tipos de invocáveis, juntamente com todos os demais tipos embutidos.

Os recursos exclusivos de Python 3 discutidos neste capítulo têm suas próprias PEPs: *PEP 3102 – Keyword-Only Arguments* (Argumentos exclusivamente nomeados, <https://www.python.org/dev/peps/pep-3102/>) e *PEP 3107 – Function Annotations* (Anotações de função, <https://www.python.org/dev/peps/pep-3107/>)

Para saber mais sobre o uso atual de anotações (meados de 2014), vale a pena ler duas perguntas de Stack Overflow: “What are good uses for Python3’s ‘Function Annotations’” (Quais são os bons usos das ‘anotações de função’ em Python 3?, <http://bit.ly/1FHiOXf>) tem uma resposta prática e comentários esclarecedores de Raymond Hettinger, e a resposta de “What good are Python function annotations?” (Para que servem as anotações de função em Python?, <http://bit.ly/1FHiN5F>) cita bastante Guido van Rossum.

Vale a pena ler a *PEP 362 – Function Signature Object* (Objeto para assinatura de função, <https://www.python.org/dev/peps/inspect2/>) se você pretende usar o módulo `inspect` que implementa esse recurso.

Uma ótima introdução à programação funcional em Python é *Functional Programming HOWTO* (HOWTO para programação funcional, <https://docs.python.org/3/howto/functional.html>) de A. M. Kuchling. O foco principal desse texto, porém, está no uso de iteradores e de geradores, que serão os assuntos do capítulo 14.

`fn.py` (<https://github.com/kachayev/fn.py>) é um pacote que dá suporte à programação funcional em Python 2 e 3. De acordo com seu autor, Alexey Kachayev, `fn.py` oferece

3 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

uma “implementação dos recursos ausentes para desfrutar da programação funcional” em Python. Esse pacote inclui um decorador `@recur.tco` que implementa otimização de tail call (chamada no final) para recursão ilimitada em Python, entre várias outras funções, estruturas de dados e receitas.

A pergunta “Python: Why is functools.partial necessary?” (Python: por que functools.partial é necessário?, <http://bit.ly/1FHiTdh>) em StackOverflow tem uma resposta extremamente informativa (e engraçada) de Alex Martelli, autor do livro clássico *Python in a Nutshell*.

Bobo de Jim Fulton provavelmente foi o primeiro framework web que mereceu ser chamado de orientado a objetos. Se ficou curioso a respeito desse framework e quiser conhecer melhor a sua versão moderna, comece pela introdução (<http://bobo.readthedocs.org/en/latest/>). Um pouco da história inicial do framework Bobo está presente em um comentário de Phillip J. Eby em uma discussão no blog de Joel Spolsky (<http://bit.ly/1FHiUxR>).

Ponto de vista

Sobre o framework Bobo

Devo minha carreira em Python ao Bobo. Usei-o em meu primeiro projeto web em Python em 1998. Descobri o framework Bobo quando procurava uma maneira orientada a objetos para criar aplicações web após ter experimentado as alternativas em Perl e Java.

Em 1997, o framework Bobo era pioneiro no conceito de publicação de objetos: mapeamento direto de URLs para uma hierarquia de objetos, sem a necessidade de configurar rotas. Fui fisgado quando vi a beleza disso. O framework Bobo também tinha tratamento automático de query HTTP baseado na análise de assinaturas de métodos ou de funções usados para tratar requisições.

Esse framework foi criado por Jim Fulton, conhecido como “The Zope Pope” (o papa do Zope), graças ao seu papel de liderança no desenvolvimento do framework Zope, que é a base de Plone CMS, SchoolTool, ERP5 e outros projetos Python de larga escala. Jim também é o criador de ZODB – o Zope Object Database (Banco de dados de objetos Zope) –, um banco de dados transacional orientado a objetos que oferece as propriedades ACID (atomicidade, consistência, isolamento e durabilidade), projetado para ser facilmente usado com Python.

Desde então, Jim reescreveu o Bobo do zero para dar suporte a WSGI e ao Python moderno (incluindo Python 3). Atualmente (quando escrevi este livro), o framework Bobo usa a biblioteca six para realizar a introspecção de funções e ser compatível com Python 2 e Python 3, apesar das mudanças na estrutura dos objetos-função e nas APIs relacionadas.

Python é uma linguagem funcional?

Por volta do ano 2000, eu estava fazendo um treinamento nos Estados Unidos quando Guido van Rossum apareceu na sala de aula (ele não era o professor). Na sessão de perguntas e respostas que se seguiu, alguém perguntou a ele quais recursos de Python foram emprestados de outras linguagens. Sua resposta: “Tudo que há de bom em Python foi roubado de outras linguagens.”

Shriram Krishnamurthi, professor de Ciência da Computação na Brown University, começa assim seu artigo “Teaching Programming Languages in a Post-Linnaean Age” (Ensinando linguagens de programação em uma era pós-Lineu, <http://bit.ly/1FHj4p2>):

Os “paradigmas” das linguagens de programação são um legado moribundo e tedioso de uma era que já se foi. Os designers das linguagens modernas não têm nenhum respeito por esses paradigmas, então por que nossos cursos estão tão servilmente presos a eles?

Nesse artigo, o nome Python é mencionado nesta passagem:

O que mais podemos dizer de uma linguagem como Python, Ruby ou Perl? Seus designers não têm nenhuma paciência com as minúcias dessas hierarquias da era de Lineu; eles emprestam recursos como querem, criando misturas que desafiam totalmente a caracterização.

Krishnamurthi aceita o fato de que, em vez de tentar classificar as linguagens com alguma taxonomia, é mais útil considerá-las como agregações de recursos. Apesar de não ter sido o objetivo de Guido, dotar a linguagem Python com funções de primeira classe abriu a porta para a programação funcional. Em seu post “Origins of Python’s *Functional* Features” (Origens dos recursos *funcionais* de Python, <http://bit.ly/1FHfhIo>), ele afirma que `map`, `filter` e `reduce` foram a motivação para acrescentar `lambda` em Python, em primeiro lugar. Todos esses recursos foram colaborações de Amrit Prem para Python 1.0 em 1994 [de acordo com Misc/HISTORY (<http://hg.python.org/cpython/file/default/Misc/HISTORY>) no código-fonte de CPython]. `lambda`, `map`, `filter` e `reduce` surgiram inicialmente em Lisp, que foi a linguagem funcional original. Entretanto Lisp não limita o que pode ser feito em um `lambda`, pois tudo em Lisp é uma expressão. A linguagem Python usa uma sintaxe orientada a comandos em que as expressões não podem conter comandos, mas muitas construções da linguagem são comandos – incluindo `try/catch`, que é do que mais sinto falta quando escrevo `lambdas`. Esse é o preço a pagar pela sintaxe altamente legível de Python.⁴ A linguagem Lisp tem muitos pontos fortes, porém a legibilidade não é um deles.

4 Há também o problema de perda de indentação ao colar códigos em fóruns na Web, mas este é outro assunto.

Ironicamente, roubar a sintaxe de list comprehension de outra linguagem funcional – a linguagem Haskell – diminuiu significativamente a necessidade de `map` e `filter`, assim como de `lambda`.

Além da sintaxe limitada das funções anônimas, o maior obstáculo para ampliar a adoção de padrões de programação funcional em Python é a falta de eliminação de recursão de cauda (tail-recursion elimination) – uma otimização que permite um processamento eficiente em termos de memória para uma função que faça uma chamada recursiva no final (tail) de seu corpo. Em outro post de blog, “Tail Recursion Elimination” (Eliminação de recursão de cauda, <http://bit.ly/1FHjdZv>), Guido apresenta diversos motivos pelos quais essa otimização não combina com Python. Esse post é uma ótima leitura para ver os argumentos técnicos, mas é melhor ainda porque os três primeiros e mais importantes motivos apresentados referem-se a problemas de usabilidade. Não é por acaso que é um prazer usar, aprender e ensinar Python. Guido a fez assim.

Então é isso: a linguagem Python, por design, não é uma linguagem funcional – seja lá o que isso queira dizer. A linguagem Python simplesmente empresta algumas ideias boas das linguagens funcionais.

O problema com as funções anônimas

Além das restrições de sintaxe específicas de Python, as funções anônimas têm uma séria desvantagem em todas as linguagens: elas não têm nome.

Estou brincando, mas não totalmente. Os `tracebacks` (listagem da pilha de execução) são mais fáceis de ler quando as funções têm nomes. As funções anônimas são um atalho prático, as pessoas se divertem programando com elas, mas, às vezes, elas se deixam levar – especialmente se a linguagem e o ambiente incentivam um aninhamento profundo de funções anônimas, como JavaScript em Node.js. Muitas funções anônimas aninhadas dificultam a depuração e o tratamento de erros. A programação assíncrona em Python é mais estruturada, talvez porque a limitação de `lambda` exija isso. Prometo escrever mais sobre programação assíncrona no futuro, mas esse assunto deverá ser adiado até o capítulo 18. A propósito, promessas, futuros e adiamentos (*deferreds*) são conceitos usados em APIs assíncronas modernas. Juntamente com as corrotinas, eles oferecem uma saída para o chamado “inferno de callbacks”. Veremos como a programação assíncrona sem `callbacks` funciona na seção “De `callbacks` a `futures` e corrotinas” na página 620.

CAPÍTULO 6

Padrões de projeto com funções de primeira classe

Conformidade aos padrões não é uma medida de qualidade.¹

— Ralph Johnson

Coautor do clássico *Design Patterns*

Apesar de os padrões de projeto serem independentes de linguagem, isso não quer dizer que todo padrão possa ser aplicado a todas as linguagens. Em sua apresentação de 1996, “Design Patterns in Dynamic Languages” (Padrões de projeto em linguagens dinâmicas, <http://norvig.com/design-patterns/>), Peter Norvig afirma que 16 dos 23 padrões do livro *Design Patterns*² original de Gamma et al. tornam-se “invisíveis ou mais simples” em uma linguagem dinâmica (slide 9). Ele estava falando de Lisp e Dylan, porém muitos dos recursos dinâmicos relevantes também estão presentes em Python.

Os autores de *Design Patterns* reconhecem na Introdução que a linguagem de implementação determina quais padrões são relevantes:

A escolha da linguagem de programação é importante porque influencia o ponto de vista de uma pessoa. Nossos padrões pressupõem recursos de linguagens no nível de Smalltalk/C++, e essa escolha determina o que pode ou não ser implementado facilmente. Se tivéssemos pressuposto o uso de linguagens procedurais, poderíamos ter incluído padrões de projeto chamados “Inheritance” (Herança), “Encapsulation” (Encapsulamento) e “Polimorphism” (Polimorfismo). De modo semelhante, alguns de nossos padrões têm suporte direto em linguagens orientadas a objetos menos comuns. A linguagem CLOS, por exemplo, tem multimétodos, que reduzem a necessidade de um padrão como Visitor (Visitante).³

1 De um slide da apresentação “Root Cause Analysis of Some Faults in Design Patterns” (Análise de causa-raiz de algumas falhas em padrões de projeto) de Ralph Johnson no IME/CCSL, Universidade de São Paulo, 15 de novembro de 2014.

2 N.T.: Edição brasileira publicada com o título “Padrões de projeto” (Bookman).

3 Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995), p. 4.

Em particular, no contexto de linguagens com funções de primeira classe, Norvig sugere repensar os padrões Strategy (Estratégia), Command (Comando), Template Method (Método template) e Visitor (Visitante). A ideia geral é que podemos substituir instâncias de algumas classes participantes nesses padrões por funções simples, reduzindo muito código repetitivo. Neste capítulo, vamos refatorar Strategy usando objetos função e discutiremos uma abordagem semelhante para simplificar o padrão Command.

Estudo de caso: refatorando Strategy

O padrão Strategy (Estratégia) é um bom exemplo de um padrão de projeto que pode ser mais simples em Python se você aproveitar as funções como objetos de primeira classe. Na seção a seguir, descreveremos e implementaremos o padrão Strategy usando a estrutura “clássica” descrita em *Design Patterns*. Se você tem familiaridade com o padrão clássico, poderá pular para a seção “Strategy orientado a função” na página 210, em que vamos refatorar o código usando funções, o que reduzirá significativamente o número de linhas.

Strategy clássico

O diagrama de classes UML da figura 6.1 representa uma organização de classes que mostra um exemplo do padrão Strategy.

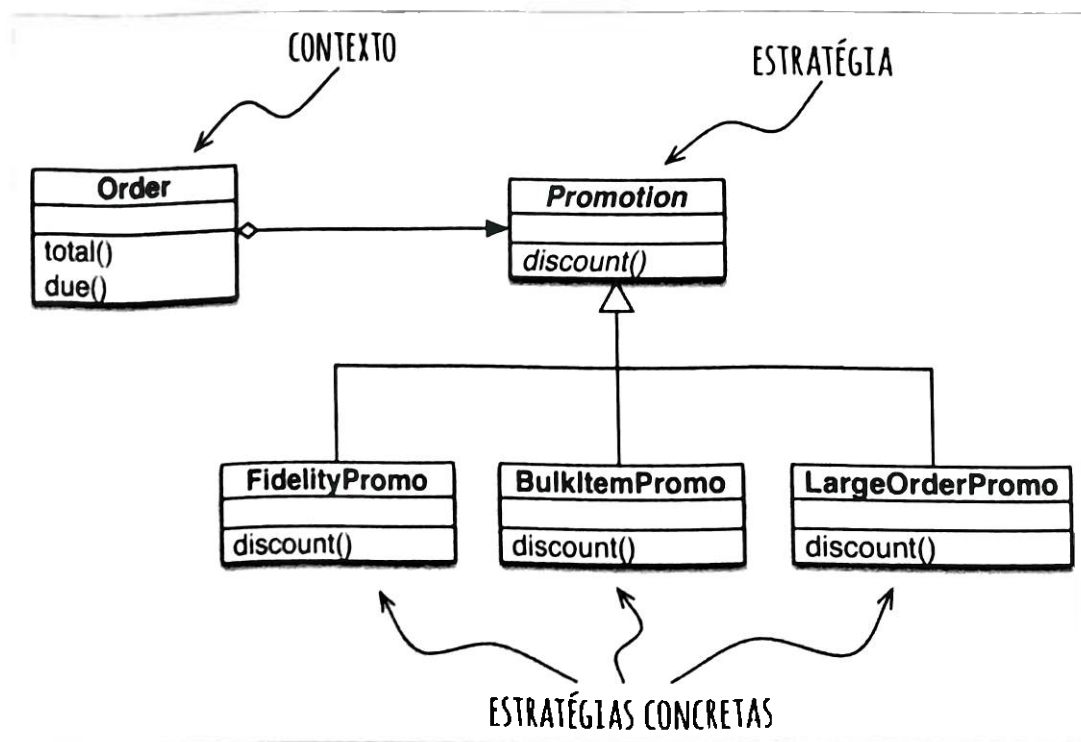


Figura 6.1 – Diagrama de classes UML para o processamento de descontos em pedidos, implementado com o padrão de projeto Strategy.

O padrão Strategy é resumido desta maneira no livro *Design Patterns*:

Define uma família de algoritmos, encapsula cada um e torna-os intercambiáveis. O padrão Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

Um exemplo claro do padrão Strategy aplicado no domínio de e-commerce está no cálculo de descontos em pedidos de acordo com os atributos do cliente ou da inspeção dos itens sendo comprados.

Considere uma loja online com as regras de desconto a seguir:

- Clientes com mil ou mais pontos no programa de fidelidade obtêm um desconto global de 5% sobre o pedido.
- Um desconto de 10% é aplicado a cada item com 20 ou mais unidades no mesmo pedido.
- Pedidos com pelo menos dez itens diferentes recebem um desconto global de 7%.

Por questões de concisão, vamos supor que somente um desconto possa ser aplicado a um pedido.

O diagrama de classes UML do padrão Strategy está representado na figura 6.1. Seus participantes são:

Contexto

Oferece um serviço delegando alguns cálculos para componentes intercambiáveis que implementam algoritmos alternativos. No exemplo de e-commerce, o contexto é um *Order* (Pedido), configurado para aplicar um desconto promocional de acordo com um dos diversos algoritmos.

Estratégia

É a interface comum aos componentes que implementam os diferentes algoritmos. Em nosso exemplo, esse papel é desempenhado por uma classe abstrata chamada *Promotion* (Promoção).

Estratégia concreta

É uma das subclasses concretas da Estratégia. *FidelityPromo*, *BulkPromo* e *LargeOrderPromo* são as três estratégias concretas implementadas.

O código do exemplo 6.1 segue o esquema da figura 6.1. Conforme descrito no livro *Design Patterns*, a estratégia concreta é escolhida pelo cliente da classe de contexto. Em nosso exemplo, antes de instanciar um pedido, o sistema, de algum modo, seleciona uma estratégia de desconto promocional e passa-a para o construtor de *Order*. A seleção da estratégia está fora do escopo do padrão.

Exemplo 6.1 – Implementação da classe Order com estratégias de desconto intercambiáveis

```
from abc import ABC, abstractmethod
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')

class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity

class Order: # o Contexto

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        fnt = '<Order total: {:.2f} due: {:.2f}>'
        return fnt.format(self.total(), self.due())

class Promotion(ABC): # a Estratégia: uma classe-base abstrata

    @abstractmethod
    def discount(self, order):
        """Devolve o desconto como um valor positivo em dólares"""
```



```

class FidelityPromo(Promotion): # primeira Estratégia Concreta
    """5% de desconto para clientes com mil ou mais pontos no programa de fidelidade"""
    def discount(self, order):
        return order.total() * .05 if order.customer.fidelity >= 1000 else 0

class BulkItemPromo(Promotion): # segunda Estratégia Concreta
    """10% de desconto para cada LineItem com 20 ou mais unidades"""
    def discount(self, order):
        discount = 0
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * .1
        return discount

class LargeOrderPromo(Promotion): # terceira Estratégia Concreta
    """7% de desconto para pedidos com 10 ou mais itens diferentes"""
    def discount(self, order):
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * .07
        return 0

```

Observe que, no exemplo 6.1, escrevi `Promotion` como uma classe-base abstrata (ABC) para poder usar o decorador `@abstractmethod`, deixando assim o padrão mais explícito.



Em Python 3.4, a maneira mais simples de declarar uma ABC é criar uma subclasse de `abc.ABC`, como fiz no exemplo 6.1. De Python 3.0 a 3.3, você deve usar a palavra reservada `metaclass=` no comando `class` (por exemplo, `class Promotion(metaclass=ABCMeta):`).

O exemplo 6.2 exhibe os doctests usados para mostrar e conferir o funcionamento de um módulo que implementa as regras descritas anteriormente.

Exemplo 6.2 – Exemplo de uso da classe `Order` com a aplicação de diferentes promoções

```

>>> joe = Customer('John Doe', 0) ❶
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, .5), ❷
...         LineItem('apple', 10, 1.5),
...         LineItem('watermellon', 5, 5.0)]
>>> Order(joe, cart, FidelityPromo()) ❸
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, FidelityPromo()) ❹

```

```

<Order total: 42.00 due: 39.90>
>>> banana_cart = [LineItem('banana', 30, .5), ❸
...                 LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, BulkItemPromo()) ❹
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0) ❺
...               for item_code in range(10)]
>>> Order(joe, long_order, LargeOrderPromo()) ❻
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, LargeOrderPromo())
<Order total: 42.00 due: 42.00>

```

- ❶ Dois clientes: joe não tem nenhum ponto no programa de fidelidade, ann tem 1.100 pontos.
- ❷ Um carrinho de compras com três itens.
- ❸ A promoção FidelityPromo não dá nenhum desconto a joe.
- ❹ ann obtém um desconto de 5% porque ela tem pelo menos 1.000 pontos.
- ❺ banana_cart tem 30 unidades do produto "banana" e 10 maçãs.
- ❻ Graças a BulkItemPromo, joe obtém um desconto de 1,50 dólar nas bananas.
- ❼ long_order tem 10 itens diferentes a um dólar cada.
- ❽ joe obtém um desconto de 7% no total do pedido por causa de LargerOrderPromo.

O exemplo 6.1 funciona perfeitamente, porém a mesma funcionalidade pode ser implementada com menos código em Python se usarmos funções como objetos. A próxima seção mostrará como isso é feito.

Strategy orientado a função

Cada estratégia concreta no exemplo 6.1 é uma classe com um único método, `discount`. Além do mais, as instâncias de estratégia não têm estado (nenhum atributo de instância). Você poderia dizer que elas se parecem muito com funções simples, e estará correto. O exemplo 6.3 apresenta uma refatoração do exemplo 6.1, em que as estratégias concretas são substituídas por funções simples e a classe abstrata `Promo` foi removida.

Exemplo 6.3 – Classe `Order` com estratégias de desconto implementadas como funções

```

from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')

```

```
class LineItem:
```

```
    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity
```

```
class Order: # o Contexto
```

```
    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion(self) ❶
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())
```

❷

```
def fidelity_promo(order): ❸
```

```
    """5% de desconto para clientes com mil ou mais pontos no programa de fidelidade"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0
```

```
def bulk_item_promo(order):
```

```
    """10% de desconto para cada LineItem com 20 ou mais unidades"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount
```

```
def large_order_promo(order):
    """7% de desconto para pedidos com 10 ou mais itens diferentes"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0
```

- ❶ Para calcular um desconto, basta chamar a função `self.promotion()`.
- ❷ Não há classe abstrata.
- ❸ Cada estratégia é uma função.

O código do exemplo 6.3 tem 12 linhas a menos que o exemplo 6.1. Usar o novo `Order` também é um pouco mais simples, conforme mostra os doctests do exemplo 6.4.

Exemplo 6.4 – Exemplo de uso da classe `Order` com as promoções como funções

```
>>> joe = Customer('John Doe', 0) ❶
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, .5),
...         LineItem('apple', 10, 1.5),
...         LineItem('watermellon', 5, 5.0)]
>>> Order(joe, cart, fidelity_promo) ❷
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, fidelity_promo)
<Order total: 42.00 due: 39.90>
3. See page 323 of Design Patterns.
4. idem, p. 196
>>> banana_cart = [LineItem('banana', 30, .5),
...                 LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, bulk_item_promo) ❸
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0)
...               for item_code in range(10)]
>>> Order(joe, long_order, large_order_promo)
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo)
<Order total: 42.00 due: 42.00>
```

- ❶ Mesma configuração de teste do exemplo 6.1.
- ❷ Para aplicar uma estratégia de desconto em um `Order`, basta passar a função de promoção como argumento.
- ❸ Uma função diferente de promoção é usada nesse caso e no próximo teste.

Observe os comentários numerados no exemplo 6.4: não há necessidade de instanciar um novo objeto promoção a cada novo pedido, pois as funções estão prontas para serem usadas.

É interessante observar que, no livro *Design Patterns*, os autores sugerem o seguinte: “Os objetos Strategy geralmente são ótimos flyweights.”⁴ Uma definição de Flyweight em outra parte dessa obra afirma que: “Um flyweight é um objeto compartilhado que pode ser usado em diversos contextos simultaneamente.”⁵ O compartilhamento é recomendado para reduzir o custo de criar um novo objeto estratégia concreto quando a mesma estratégia é aplicada repetidamente a cada novo contexto – a cada nova instância de *Order*, em nosso exemplo. Sendo assim, para contornar uma desvantagem do padrão Strategy – seu custo de tempo de execução –, os autores recomendaram aplicar outro padrão. Enquanto isso, o número de linhas e o custo de manutenção de seu código aumentam.

Um caso de uso mais complicado, com estratégias concretas complexas que armazenem estados internos, pode exigir todas as partes dos padrões de projeto Strategy e Flyweight combinadas. Com frequência, porém, as estratégias concretas não têm estados internos; elas somente lidam com dados do contexto. Se esse for o caso, definitivamente, use as boas e velhas funções simples em vez de escrever classes com métodos únicos que implementem uma interface com um único método, declarada em outra classe. Uma função é mais leve que uma instância de uma classe definida pelo usuário, e não há necessidade de usar Flyweight, pois cada função de estratégia será criada somente uma vez por Python quando ele compilar o módulo. Uma função simples também é “um objeto compartilhado que pode ser usado em diversos contextos simultaneamente”.

Agora que implementamos o padrão Strategy com funções, outras possibilidades emergem. Suponha que você queira criar uma “metaestratégia” que selecione os melhores descontos disponíveis para um dado pedido. Nas seções a seguir, apresentamos refatorações adicionais que implementam esse requisito usando uma variedade de abordagens que aproveitam funções e módulos como objetos.

Escolhendo a melhor estratégia: abordagem simples

Dados os mesmos clientes e carrinhos de compra dos testes do exemplo 6.4, vamos agora adicionar três testes no exemplo 6.5.

Exemplo 6.5 – A função `best_promo` aplica todos os descontos e retorna o maior

```
>>> Order(joe, long_order, best_promo) ❶  
<Order total: 10.00 due: 9.30>
```

⁴ Veja a página 323 do livro *Design Patterns*.

⁵ *Idem*, p. 196

```
>>> Order(joe, banana_cart, best_promo) ❷
<Order total: 30.00 due: 28.50>
>>> Order(ann, cart, best_promo) ❸
<Order total: 42.00 due: 39.90>
```

- ❶ best_promo selecionou larger_order_promo para o cliente joe.
- ❷ Nesse caso, joe conseguiu o desconto bulk_item_promo por comprar muitas bananas.
- ❸ No checkout com um carrinho simples, best_promo deu à fiel cliente ann o desconto de fidelity_promo.

A implementação de best_promo é bem simples. Veja o exemplo 6.6.

Exemplo 6.6 – best_promo encontra o desconto máximo fazendo uma iteração por uma lista de funções

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo] ❶

def best_promo(order): ❷
    """Seleciona o melhor desconto disponível
    """
    return max(promo(order) for promo in promos) ❸
```

- ❶ promos: lista de estratégias implementadas como funções.
- ❷ best_promo aceita uma instância de Order como argumento, assim como as outras funções *_promo.
- ❸ Usando uma expressão geradora, aplicamos cada uma das funções de promos a order e devolvemos o desconto máximo calculado.

O exemplo 6.6 é simples: promos é uma list de funções. Uma vez que você se acostuma com a ideia de que as funções são objetos de primeira classe, é natural criar estruturas de dados que armazenem funções quando isso ajuda a resolver um problema.

Apesar de o exemplo 6.6 funcionar e ser fácil de ler, há algumas duplicações que poderiam resultar em um bug sutil: para acrescentar uma nova estratégia de promoção, precisamos escrever a função e lembrar de adicioná-la à lista promos; caso contrário, a nova promoção funcionará quando passada explicitamente como argumento a Order, mas não será levada em consideração por best_promotion.

Continue lendo para ver algumas soluções para esse problema.

Encontrando estratégias em um módulo

Módulos em Python também são objetos de primeira classe, e a biblioteca-padrão disponibiliza várias funções para tratá-los. A função embutida `globals` está descrita da seguinte maneira na documentação de Python:

`globals()`

Devolve um dicionário que representa a tabela atual de símbolos globais. É sempre o dicionário do módulo atual (em uma função ou método, é o módulo em que ele está definido, e não o módulo a partir do qual ele é chamado).

O exemplo 6.7 é quase um hack para usar `globals` a fim de ajudar `best_promo` a encontrar automaticamente as outras funções `*_promo` disponíveis.

Exemplo 6.7 – A lista `promos` é criada por introspecção do namespace global do módulo

```
promos = [globals()[name] for name in globals() ❶
          if name.endswith('_promo') ❷
          and name != 'best_promo'] ❸

def best_promo(order):
    """Seleciona o melhor desconto disponível
    """
    return max(promo(order) for promo in promos) ❹
```

- ❶ Faz a iteração em cada `name` no dicionário devolvido por `globals()`.
- ❷ Seleciona somente os nomes que terminem com o sufixo `_promo`.
- ❸ Filtra a própria `best_promo` para evitar uma recursão infinita.
- ❹ Nenhuma alteração em `best_promo`.

Outra maneira de reunir as promoções disponíveis é criar um módulo e colocar todas as funções de estratégia aí, exceto `best_promo`.

No exemplo 6.8, a única alteração significativa é que a lista de funções de estratégia é criada por introspecção em um módulo diferente chamado `promotions`. Observe que o exemplo 6.8 depende da importação do módulo `promotions` e também de `inspect`, que oferece funções de introspecção de alto nível (as importações não foram mostradas por questão de concisão, pois elas normalmente estariam no início do arquivo).

Exemplo 6.8 – A lista `promos` é criada por introspecção em um novo módulo `promotions`

```
promos = [func for name, func in
          inspect.getmembers(promotions, inspect.isfunction)]

def best_promo(order):
    """Seleciona o melhor desconto disponível
    """
    return max(promo(order) for promo in promos)
```


A função `inspect.getmembers` devolve os atributos de um objeto – nesse caso, o módulo `promotions` – opcionalmente filtrado por um predicado (uma função booleana). Usamos `inspect.isfunction` para obter somente as funções do módulo.

O exemplo 6.8 funciona independentemente dos nomes dados às funções; tudo que importa é que o módulo `promotions` contenha somente funções que calculem os descontos concedidos aos pedidos. É claro que essa é uma suposição implícita do código. Se alguém criasse uma função com uma assinatura diferente no módulo `promotions`, `best_promo` provocaria um erro ao tentar aplicá-la a um pedido.

Poderíamos acrescentar testes mais rigorosos para filtrar as funções inspecionando seus argumentos, por exemplo. A questão principal no exemplo 6.8 não é oferecer uma solução completa, mas destacar um uso possível de introspecção de módulo.

Uma alternativa mais explícita para reunir funções de descontos promocionais dinamicamente seria usar um decorador simples. Mostraremos outra versão de nosso exemplo de Strategy aplicado a e-commerce no capítulo 7, que lidará com decoradores de funções.

Na próxima seção, discutiremos o padrão Command (Comando) – outro padrão de projeto que, às vezes, é implementado com classes de métodos únicos quando funções simples bastariam.

Command

Command é outro padrão de projeto que pode ser simplificado pelo uso de funções passadas como argumentos. A figura 6.2 mostra a organização das classes no padrão Command.

O objetivo de Command é desacoplar um objeto que chama uma operação (o Chamador) do objeto provedor que a implementa (o Receptor). No exemplo do livro *Design Patterns*, cada chamador é um item de menu em uma aplicação gráfica, e os receptores são o documento editado ou a própria aplicação.

A ideia é colocar um objeto `Command` entre os dois, implementando uma interface com um único método `execute`, que chama algum método no Receptor para executar a operação desejada. Dessa maneira, o Chamador não precisará conhecer a interface do Receptor e diferentes receptores poderão ser adaptados por meio de subclasses diferentes de `Command`. O Chamador é configurado com um comando concreto e chama o seu método `execute` para operá-lo. Na figura 6.2, observe que `MacroCommand` pode armazenar uma sequência de comandos; seu método `execute()` chama o mesmo método em cada comando armazenado.

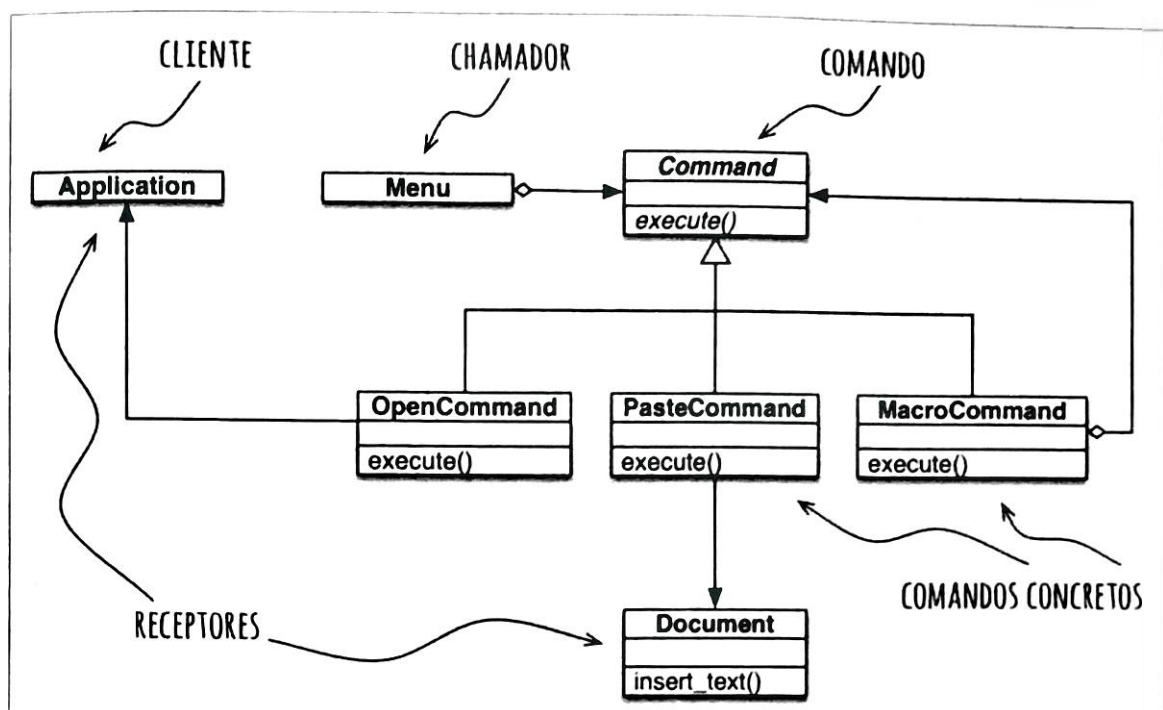


Figura 6.2 – Diagrama de classes UML para um editor de texto orientado a menus implementado com o padrão de projeto Command. Cada comando pode ter um receptor diferente: o objeto que implementa a ação. Para PasteCommand, o receptor é Document. Para OpenCommand, o receptor é a aplicação.

Citando Gamma et al., “Commands são um substituto orientado a objetos para callbacks.” A questão é: precisamos de um substituto orientado a objetos para callbacks? Às vezes, sim, mas nem sempre.

Em vez de fornecer uma instância de Command ao Chamador, podemos simplesmente fornecer-lhe uma função. Em vez de chamar `command.execute()`, o Chamador poderá simplesmente chamar `command()`. MacroCommand pode ser implementado com uma classe que implemente `__call__`. As instâncias de MacroCommand seriam invocáveis (callable), cada uma armazenando uma lista de funções para chamadas futuras, conforme implementado no exemplo 6.9.

Exemplo 6.9 – Cada instância de MacroCommand tem uma lista interna de comandos

class MacroCommand:

"""Um comando que executa uma lista de comandos"""

def __init__(self, commands):

self.commands = list(commands) # ❶

def __call__(self):

for command in self.commands: # ❷

command()

- ❶ Criar uma lista a partir do argumento `commands` garante que ela seja iterável e mantém uma cópia local das referências aos comandos em cada instância de MacroCommand.

- ② Quando uma instância de `MacroCommand` é chamada, cada comando em `self.commands` é chamado em sequência.

Usos mais sofisticados do padrão `Command` – para tratar `undo` (desfazer), por exemplo – podem exigir mais que uma simples função de `callback`. Mesmo assim, Python oferece algumas alternativas que merecem consideração:

- Uma instância de invocável como `MacroCommand` no exemplo 6.9 pode manter qualquer estado que seja necessário, e ainda oferecer métodos extras além de `__call__`.
- Uma `closure` pode ser usada para armazenar o estado interno de uma função entre chamadas.

Com isso, revisitamos o padrão `Command` com funções de primeira classe. Em nível geral, a abordagem, nesse caso, foi semelhante àquela aplicada ao padrão `Strategy`: substituir as instâncias de uma classe participante que implementava uma interface com um único método por invocáveis. Afinal de contas, todo invocável em Python implementa uma interface de um único método, e esse método chama-se `__call__`.

Resumo do capítulo

Como Peter Norvig destacou alguns anos depois que o livro clássico *Design Patterns* surgiu, “16 de 23 padrões têm implementação qualitativamente mais simples em Lisp ou Dylan que em C++, ao menos para alguns usos de cada padrão” [slide 9 da apresentação “Design Patterns in Dynamic Languages” (Padrões de projeto em linguagens dinâmicas, <http://bit.ly/1HGC0r5>) de Norvig]. Python compartilha alguns dos recursos dinâmicos das linguagens Lisp e Dylan, em particular as funções de primeira classe – nosso foco nesta parte do livro.

Na mesma palestra da qual a citação do início deste capítulo foi extraída, ao refletir sobre o vigésimo aniversário de *Design Patterns: Elements of Reusable Object-Oriented Software*, Ralph Johnson afirmou que uma das falhas do livro é dar “muita ênfase aos padrões como pontos finais em vez de serem passos no processo de design”.⁶ Neste capítulo, usamos o padrão `Strategy` como ponto de partida: uma solução inicial que simplificamos usando funções de primeira classe.

Em muitos casos, funções ou objetos invocáveis oferecem uma maneira mais natural de implementar `callbacks` em Python do que imitar os padrões `Strategy` e `Command` conforme descritos por Gamma, Helm, Johnson e Vlissides. A refatoração de `Strategy` e a discussão de `Command` neste capítulo são exemplos de uma percepção mais geral: às

⁶ Da mesma palestra citada no início deste capítulo: “Root Cause Analysis of Some Faults in Design Patterns” (Análise de causa-raiz de algumas falhas em padrões de projeto), apresentada por Johnson no IME-USP, em 15 de novembro de 2014.

vezes, você encontra um padrão de projeto ou uma API que exige que os componentes implementem uma interface com um único método, e esse método tem um nome que soa genérico como “execute”, “run” ou “doIt”. Esses padrões ou APIs geralmente podem ser implementados com menos código repetitivo em Python usando funções de primeira classe ou outros invocáveis.

A mensagem dos slides sobre padrões de projeto de Peter Norvig é que os padrões Command e Strategy – juntamente com Template Method e Visitor – podem ser simplificados ou até mesmo se tornar “invisíveis” com funções de primeira classe, pelo menos para algumas aplicações desses padrões.

Leituras complementares

Nossa discussão sobre Strategy terminou com uma sugestão de que decoradores de função podem ser usados para melhorar o exemplo 6.8. Também mencionamos o uso de closures algumas vezes neste capítulo. Os decoradores, assim como as closures, são o foco do capítulo 7. Aquele capítulo começa com uma refatoração do exemplo de e-commerce usando um decorador para registrar as promoções disponíveis.

A “Recipe 8.21. Implementing the Visitor Pattern” (Implementar o padrão Visitor) em *Python Cookbook*, 3ª edição (O’Reilly),⁷ de David Beazley e Brian K. Jones, apresenta uma implementação elegante do padrão Visitor em que uma classe `NodeVisitor` trata métodos como objetos de primeira classe.

Sobre o tema geral de padrões de projeto, a opção de leituras para o programador Python não é tão ampla quanto o que está disponível para comunidades de outras linguagens.

Até onde sei, *Learning Python Design Patterns* de Gennadiy Zlobin (Packt) é o único livro totalmente dedicado a padrões em Python atualmente (junho de 2014). Porém o trabalho de Zlobin é bem conciso (cem páginas) e abrange 8 dos 23 padrões de projeto originais.

Expert Python Programming de Tarek Ziadé (Packt) é um dos melhores livros de Python de nível intermediário no mercado, e seu último capítulo – “Useful Design Patterns” – apresenta sete dos padrões clássicos de um ponto de vista pythônico.

Alex Martelli fez diversas palestras sobre Padrões de Projeto em Python. Há um vídeo de sua apresentação na EuroPython 2011 (<http://bit.ly/1HGBXvx>) e um conjunto de slides em seu site pessoal (http://www.aleax.it/gdd_pydp.pdf). Encontrei diferentes conjuntos de slides e vídeos ao longo dos anos, com tamanhos variados, portanto vale a pena fazer uma pesquisa detalhada com o nome dele e as palavras “Python Design Patterns”.

7 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

Por volta de 2008, Bruce Eckel – autor do excelente livro *Thinking in Java* (Prentice Hall) – começou a escrever um livro chamado *Python 3 Patterns, Recipes and Idioms* (<http://bit.ly/1HGBXeQ>). Era para ser escrito por uma comunidade de colaboradores liderada por Eckel, porém, seis anos depois, ainda está incompleto e, aparentemente, parado (quando escrevi esta obra, a última atualização no repositório havia ocorrido dois anos antes).

Há vários livros sobre padrões de projeto no contexto de Java, mas, entre eles, aquele de que mais gosto é *Head First Design Patterns* de Eric Freeman, Bert Bates, Kathy Sierra e Elisabeth Robson (O'Reilly)⁸. O livro explica 16 dos 23 padrões clássicos. Se você gosta do estilo aloprado da série *Head First* e precisa de uma introdução a esse assunto, amará esse trabalho. Entretanto ele é centrado em Java.

Para um olhar renovado sobre os padrões do ponto de vista de uma linguagem dinâmica com duck typing (tipagem pato) e funções de primeira classe, *Design Patterns in Ruby* de Russ Olsen (Addison-Wesley) tem muitas ideias esclarecedoras que também são aplicáveis a Python. Apesar das várias diferenças sintáticas, em nível semântico, Python e Ruby são mais próximos entre si do que em relação a Java ou C++.

Em *Design Patterns in Dynamic Languages* (Padrões de projeto em linguagens dinâmicas, slides em <http://norvig.com/design-patterns/>), Peter Norvig mostra como as funções de primeira classe (e outros recursos dinâmicos) tornam vários dos padrões de projeto originais mais simples ou desnecessários.

É claro que o livro *Design Patterns* original de Gamma et al.⁹ é leitura obrigatória para levar esse assunto a sério. A Introdução por si só vale o preço. Ela é a fonte dos princípios de design citados com tanta frequência: “Programe para uma interface, e não para uma implementação” e “Favoreça a composição de objetos em vez da herança de classes”.

Ponto de vista

A linguagem Python tem funções e tipos de primeira classe, recursos que, segundo Norvig, afetam 10 dos 23 padrões [slide 10 de *Design Patterns in Dynamic Languages* (Padrões de projeto em linguagens dinâmicas, <http://norvig.com/design-patterns/>)]. No próximo capítulo, veremos que Python também tem funções genéricas (“Funções genéricas com dispatch único” na página 243), semelhante aos multimétodos de CLOS, que Gamma et al. sugerem como uma maneira mais simples de implementar o padrão clássico Visitor. Por outro lado, Norvig diz que os multimétodos simplificam o padrão Builder (Slide 10). Combinar padrões de projeto com recursos da linguagem não é uma ciência exata.

8 N.T.: Edição brasileira publicada com o título “Use a Cabeça! Padrões de projeto” (Altabooks).

9 N.T.: Edição brasileira publicada com o título “Padrões de Projeto” (Bookman).

Em salas de aula ao redor do mundo, os padrões de projeto frequentemente são ensinados com exemplos em Java. Já ouvi mais de um aluno afirmar que foi levado a acreditar que os padrões de projeto originais seriam úteis em qualquer linguagem de implementação. Acontece que os 23 padrões “clássicos” do livro de Gamma et al. aplicam-se muito bem ao Java “clássico”, apesar de serem originalmente apresentados, em sua maior parte, no contexto de C++ – alguns têm exemplos em Smalltalk no livro. Contudo isso não significa que todos os padrões sejam igualmente bem aplicados em qualquer linguagem. Os autores são explícitos logo no início do livro dizendo que “alguns de nossos padrões têm suporte direto de linguagens orientadas a objetos menos comuns” (lembre-se da citação completa na primeira página deste capítulo).

A bibliografia de Python sobre padrões de projeto é bem pequena quando comparada à de Java, C++ ou Ruby. Na seção “Leituras complementares” na página 219, mencionei o livro *Learning Python Design Patterns* de Gennadiy Zlobin, publicado recentemente em novembro de 2013. Em comparação, o livro *Design Patterns in Ruby* de Russ Olsen foi publicado em 2007 e tem 384 páginas – 284 a mais que a obra de Zlobin.

Agora que Python está se tornando cada vez mais popular entre os acadêmicos, vamos esperar que mais obras sejam escritas sobre padrões de projeto no contexto dessa linguagem. Além disso, Java 8 introduziu referências a métodos e funções anônimas, e esses recursos muito esperados provavelmente possibilitarão novas abordagens aos padrões em Java – reconhecendo que as linguagens evoluem e o mesmo deve ocorrer com o nosso entendimento sobre como aplicar os padrões de projeto clássicos.

CAPÍTULO 7

Decoradores de função e closures

Tem havido diversas reclamações sobre a escolha do nome “*decorator*” para esse recurso. A principal delas é que o nome não é consistente com seu uso no livro da GoF.¹ O nome *decorador* provavelmente tem mais a ver com seu uso na área de compiladores – uma árvore de sintaxe é percorrida e anotada.

— PEP 318 — Decorators for Functions and Methods (Decoradores para funções e métodos)

Decoradores (*decorators*) de função nos permitem “marcar” funções no código-fonte para modificar o seu comportamento de alguma maneira. É um recurso poderoso, mas dominá-lo exige compreender as closures.

Uma das palavras reservadas mais recentes em Python é `nonlocal`, introduzida em Python 3.0. Você pode ter uma vida produtiva como programador Python sem jamais usá-la se adotar uma rígida doutrina de orientação a objetos centrada em classes. No entanto, se quiser implementar seus próprios decoradores de função, você precisará conhecer muito bem as closures e, então, a necessidade de usar `nonlocal` se tornará óbvia.

Além de sua aplicação em decoradores, as closures também são essenciais para uma programação assíncrona eficiente com callbacks e para codar em estilo funcional sempre que isso fizer sentido.

O objetivo final deste capítulo é explicar exatamente como os decoradores de função operam, desde os decoradores de registro mais simples até os parametrizados, mais complicados. No entanto, antes de atingir esse objetivo, precisamos discutir:

- como Python avalia a sintaxe dos decoradores;
- como Python decide se uma variável é local;
- por que as closures existem e como funcionam;
- o problema resolvido por `nonlocal`.

¹ Trata-se do livro *Design Patterns* de 1995 da chamada Gangue dos Quatro (Gang of Four).

Com essa base, podemos atacar outros assuntos relacionados a decoradores:

- implementação de um decorador bem comportado;
- decoradores interessantes da biblioteca-padrão;
- implementação de um decorador parametrizado.

Começaremos com uma introdução básica aos decoradores e, em seguida, continuaremos com o restante dos itens listados aqui.

Básico sobre decoradores

Um decorador é um invocável (callable) que aceita outra função como argumento (a função decorada).² O decorador pode realizar algum processamento com a função decorada e devolvê-la ou substituí-la por outra função ou um objeto invocável.

Em outras palavras, supondo que haja um decorador chamado `decorate`, este código:

```
@decorate
def target():
    print('running target()')
```

tem o mesmo efeito que escrever:

```
def target():
    print('running target()')

target = decorate(target)
```

O resultado é o mesmo: no final de qualquer um desses trechos de código, o nome `target` não necessariamente fará referência à função original `target`, mas a alguma função devolvida por `decorate(target)`.

Para confirmar que a função decorada é substituída, veja a sessão de console no exemplo 7.1.

Exemplo 7.1 – Um decorador normalmente substitui uma função por outra função diferente

```
>>> def deco(func):
...     def inner():
...         print('running inner()')
...     return inner ❶
...
>>> @deco
... def target(): ❷
```

² A linguagem Python também aceita decoradores de classe, que serão discutidos no capítulo 21.

```

...     print('running target()')
...
>>> target() ❸
running inner()
>>> target ❹
<function deco.<locals>.inner at 0x10063b598>

```

- ❶ deco devolve seu objeto função inner.
- ❷ target é decorado por deco.
- ❸ Chamar a função target decorada; na verdade, executa inner.
- ❹ A inspeção revela que target agora é uma referência a inner.

Falando estritamente, os decoradores são apenas um açúcar sintático. Como acabamos de ver, você sempre pode simplesmente chamar um decorador como qualquer invocável normal, passando outra função. Às vezes, isso será conveniente, especialmente em *metaprogramação* – para alterar o comportamento do programa em tempo de execução.

Para resumir: o primeiro fato essencial sobre os decoradores é que eles têm a capacidade de substituir a função decorada por uma função diferente. O segundo fato essencial é que eles são executados imediatamente quando um módulo é carregado. Isso será explicado a seguir.

Quando Python executa os decoradores

Uma característica fundamental dos decoradores é que eles são executados imediatamente após a função decorada ser definida. Normalmente, isso ocorre em *tempo de importação* (isto é, quando um módulo é carregado pelo interpretador Python). Considere o módulo *registration.py* no exemplo 7.2.

Exemplo 7.2 – O módulo *registration.py*

```

registry = [] ❶

def register(func): ❷
    print('running register(%s)' % func) ❸
    registry.append(func) ❹
    return func ❺

@register ❻
def f1():
    print('running f1()')

@register

```

```
def f2():
    print('running f2()')

def f3(): ❷
    print('running f3()')

def main(): ❸
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main() ❹
```

- ❶ registry armazenará referências a funções decoradas com `@register`.
- ❷ register recebe uma função como argumento.
- ❸ Exibe a função que está sendo decorada, para demonstração.
- ❹ Inclui func em registry.
- ❺ Devolve func: precisamos devolver uma função; nesse caso, devolvemos a mesma função recebida como argumento.
- ❻ f1 e f2 são decoradas com `@register`.
- ❼ f3 não é decorada.
- ❽ main exibe registry e então chama f1(), f2() e f3().
- ❾ main() é chamado somente se *registration.py* executar como script.

A saída da execução de *registration.py* como script fica assim:

```
$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]
running f1()
running f2()
running f3()
```

Observe que register executa (duas vezes) antes de qualquer outra função no módulo. Quando chamada, register recebe o objeto-função sendo decorado como argumento – por exemplo, `<function f1 at 0x100631bf8>`.

Depois que o módulo é carregado, `registry` contém referências para as duas funções decoradas: `f1` e `f2`. Essas funções, assim como `f3`, são executadas somente quando chamadas explicitamente por `main`.

Se `registration.py` for importado (em vez de ser executado como script), a saída será:

```
>>> import registration
running register(<function f1 at 0x10063b1e0>)
running register(<function f2 at 0x10063b268>)
```

A essa altura, se observar `registry`, você verá o seguinte:

```
>>> registration.registry
[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]
```

O ponto principal no exemplo 7.2 é enfatizar que os decoradores de função são executados assim que o módulo é importado, porém as funções decoradas executam somente quando são explicitamente chamadas. Isso ilustra a diferença entre o que os Pythonistas chamam de *tempo de importação* (`import time`) e *tempo de execução* (`runtime`).

Considerando como os decoradores são comumente empregados em um código de verdade, o exemplo 7.2 é incomum por dois motivos:

- A função decoradora está definida no mesmo módulo que as funções decoradas. Um decorador de verdade normalmente é definido em um módulo e aplicado a funções em outros módulos.
- O decorador `register` devolve a mesma função recebida como argumento. Na prática, a maioria dos decoradores define uma função interna e a devolve.

Apesar de o decorador `register` do exemplo 7.2 devolver a função decorada inalterada, essa técnica não deixa de ser útil. Decoradores semelhantes são usados em muitos frameworks web Python para adicionar funções a algum registro centralizado – por exemplo, um registro que mapeie padrões de URL a funções que gerem respostas HTTP. Esses decoradores de registro podem ou não alterar a função decorada. A próxima seção mostra um exemplo prático.

Padrão Strategy melhorado com decorador

Um decorador de registro é uma boa melhoria para o desconto promocional no exemplo de e-commerce da seção “Estudo de caso: refatorando Strategy” na página 206.

Lembre-se de que nosso principal problema no exemplo 6.6 era a repetição dos nomes das funções em suas definições e na lista `promos`, usada pela função `best_promo` para determinar o maior desconto aplicável. A repetição é problemática, pois alguém poderia

adicionar uma nova função de estratégia promocional e esquecer-se de acrescentá-la manualmente na lista `promos` – caso em que `best_promo` ignorará silenciosamente a nova estratégia, introduzindo um bug sutil no sistema. O exemplo 7.3 resolve esse problema com um decorador de registro.

Exemplo 7.3 – A lista `promos` é preenchida pelo decorador `promotion`

```
promos = [] ❶

def promotion(promo_func): ❷
    promos.append(promo_func)
    return promo_func

@promotion ❸
def fidelity(order):
    """5% de desconto para clientes com mil ou mais pontos no programa de fidelidade"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0

@promotion
def bulk_item(order):
    """10% de desconto para cada LineItem com 20 ou mais unidades"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount

@promotion
def large_order(order):
    """7% de desconto para pedidos com 10 ou mais itens diferentes"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0

def best_promo(order): ❹
    """Seleciona o melhor desconto disponível"""
    return max(promo(order) for promo in promos)
```

- ❶ A lista `promos` começa vazia.
- ❷ O decorador `promotion` devolve `promo_func` inalterada após adicioná-la à lista `promos`.
- ❸ Qualquer função decorada com `@promotion` será adicionada a `promos`.
- ❹ Nenhum alteração é necessária em `best_promos`, pois ela baseia-se na lista `promos`.

Essa solução tem diversas vantagens sobre as demais apresentadas na seção “Estudo de caso: refatorando Strategy” na página 206:

- As funções de estratégia de promoção não precisam usar nomes especiais (isto é, elas não precisam usar o sufixo `_promo`).
- O decorador `@promotion` enfatiza o propósito da função decorada, além de tornar mais fácil desabilitar uma promoção temporariamente: basta comentar o decorador para desativá-lo.
- As estratégias de descontos promocionais podem ser definidas em outros módulos em qualquer lugar no sistema, desde que o decorador `@promotion` seja aplicado a elas.

A maioria dos decoradores altera a função decorada. Esses decoradores normalmente fazem isso definindo uma função interna e devolvendo-a para substituir a função decorada. Códigos que usam funções internas quase sempre dependem de closures para funcionar corretamente. Para entender as closures, devemos dar um passo para trás e analisar mais de perto o funcionamento dos escopos de variáveis em Python.

Regras para escopo de variáveis

No exemplo 7.4, definimos e testamos uma função que lê duas variáveis: uma variável local `a`, definida como parâmetro da função, e uma variável `b`, que não está definida em nenhum lugar na função.

Exemplo 7.4 – Função que lê uma variável local e uma variável global

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

O erro que vimos não causa nenhuma surpresa. Prosseguindo com o exemplo 7.4, se atribuirmos um valor a uma variável global `b` e, em seguida, chamarmos `f1`, o código funciona:

```
>>> b = 6
>>> f1(3)
3
6
```

Agora vamos ver um exemplo que poderá surpreender você.

Dê uma olhada na função `f2` do exemplo 7.5. Suas duas primeiras linhas são iguais às da função `f1` do exemplo 7.4 e, em seguida, `f2` atribui um valor a `b` e exibe o valor. Porém ela falha no segundo `print`, antes de a atribuição ser feita.

Exemplo 7.5 – Variável `b` é local porque ela recebe um valor no corpo da função

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```

Observe que a saída começa com 3, o que prova que o comando `print(a)` foi executado. Porém o segundo comando, `print(b)`, não é executado. Quando vi isso pela primeira vez, fiquei surpreso, achando que 6 deveria ser exibido, pois há uma variável global `b` e a atribuição à variável `b` local é feita após `print(b)`.

Mas o fato é que, quando compila o corpo da função, o interpretador Python decide que `b` é uma variável local, pois sua atribuição é feita dentro da função. O bytecode gerado reflete essa decisão e tenta acessar `b` no ambiente local. Posteriormente, quando a chamada a `f2(3)` é feita, o corpo de `f2` busca e exibe o valor da variável local `a`, mas ao tentar buscar o valor da variável local `b`, ela descobre que `b` não tem associação (*unbound* é o termo em inglês para “sem associação”).

Não é um bug, mas uma opção de design: Python não exige que você declare variáveis, mas supõe que uma variável cujo valor tenha sido atribuído no corpo de uma função é local. Isso é muito melhor que o comportamento de JavaScript, que também não exige declarações de variáveis, mas se esquecer de declarar que uma variável é local (com `var`), você poderá sobrescrever uma variável global sem saber.

Se quisermos que o interpretador trate `b` como uma variável global apesar da atribuição dentro da função, devemos usar a declaração `global`:

```
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...
>>> f3(3)
3
6
>>> b
9

>>> f3(3)
a = 3
b = 8
b = 30
>>> b
30
>>>
```

Depois de termos visto mais de perto como os escopos de variáveis funcionam em Python, podemos atacar as closures na próxima seção – “Closures” – na página 232. Se você estiver curioso a respeito das diferenças entre os bytecodes das funções nos exemplos 7.4 e 7.5, consulte a caixa de texto a seguir.

Comparando bytecodes

O módulo `dis` oferece uma maneira fácil de fazer o *disassembly* do bytecode de funções Python. Veja os exemplos 7.6 e 7.7, que mostram os bytecodes de `f1` e `f2` dos exemplos 7.4 e 7.5.

Exemplo 7.6 – Disassembly da função `f1` do exemplo 7.4

```
>>> from dis import dis
>>> dis(f1)
2          0 LOAD_GLOBAL               0 (print) ❶
          3 LOAD_FAST                0 (a) ❷
          6 CALL_FUNCTION           1 (1 positional, 0 keyword pair)
          9 POP_TOP
```

3	10 LOAD_GLOBAL	0 (print)
	13 LOAD_GLOBAL	1 (b) ❸
	16 CALL_FUNCTION	1 (1 positional, 0 keyword pair)
	19 POP_TOP	
	20 LOAD_CONST	0 (None)
	23 RETURN_VALUE	

- ❶ Carrega o nome global `print`.
- ❷ Carrega o nome local `a`.
- ❸ Carrega o nome global `b`.

Compare o bytecode de `f1` exibido no exemplo 7.6 com o bytecode de `f2` no exemplo 7.7.

Exemplo 7.7 – Disassembly da função `f2` do exemplo 7.5

```
>>> dis(f2)
```

2	0 LOAD_GLOBAL	0 (print)
	3 LOAD_FAST	0 (a)
	6 CALL_FUNCTION	1 (1 positional, 0 keyword pair)
	9 POP_TOP	
3	10 LOAD_GLOBAL	0 (print)
	13 LOAD_FAST	1 (b) ❶
	16 CALL_FUNCTION	1 (1 positional, 0 keyword pair)
	19 POP_TOP	
4	20 LOAD_CONST	1 (9)
	23 STORE_FAST	1 (b)
	26 LOAD_CONST	0 (None)
	29 RETURN_VALUE	

- ❶ Carrega o nome *local* `b`. Isso mostra que o compilador considera `b` uma variável local, mesmo que a atribuição a `b` ocorra posteriormente, pois a natureza da variável – o fato de ser local ou não – não pode alterar o corpo da função.

A máquina virtual de CPython que executa o bytecode é uma stack machine, portanto as operações `LOAD` e `POP` referem-se à pilha. Uma descrição mais detalhada dos opcodes Python está além do escopo deste livro, porém eles estão documentados junto ao módulo `dis` em `dis – Disassembler for Python bytecode` (`dis` – disassembler para bytecode Python, <http://docs.python.org/3/library/dis.html>).

Closures

Na blogosfera, as closures às vezes são confundidas com funções anônimas. O motivo pelo qual muitos as confundem é histórico: definir funções dentro de funções não é tão comum até você começar a usar funções anônimas. E as closures são importantes somente quando você tem funções aninhadas. Desse modo, muitas pessoas aprendem os dois conceitos ao mesmo tempo.

Na verdade, uma closure é uma função com um escopo estendido, que engloba variáveis não globais referenciadas no corpo da função que não estão definidas ali. Não importa se a função é anônima ou não; o importante é ela poder acessar variáveis não globais definidas fora de seu corpo.

Esse é um conceito desafiador, difícil de dominar, e será melhor abordá-lo por meio de um exemplo.

Considere uma função `avg` que calcula a média de uma série de valores sempre crescente; por exemplo, o preço de fechamento médio de uma commodity em toda a sua história. Todos os dias, um novo preço é adicionado e a média é calculada levando em conta todos os preços até então.

Partindo de um histórico limpo, é assim que `avg` poderia ser usada:

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

De onde vem `avg` e onde ela mantém o histórico dos valores anteriores?

Para começar, o exemplo 7.8 é uma implementação baseada em classe.

Exemplo 7.8 – `average_oo.py`: uma classe para calcular uma média em evolução

```
class Averager():
    def __init__(self):
        self.series = []
    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

A classe `Averager` cria instâncias que são invocáveis:

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

O exemplo 7.9 é uma implementação funcional que usa a função de ordem superior `make_averager`.

Exemplo 7.9 – `average.py`: uma função de ordem superior para calcular uma média em evolução

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

Quando chamada, `make_averager` devolve um objeto função `averager`. Sempre que um `averager` é chamado, ele concatena o argumento passado à série e calcula a média atual, como mostra o exemplo 7.10.

Exemplo 7.10 – Testando o exemplo 7.9.

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Observe as semelhanças entre os exemplos: chamamos `Averager()` ou `make_averager()` para obter um objeto invocável `avg` que atualizará a série histórica e calculará a média atual. No exemplo 7.8, `avg` é uma instância de `Averager` e, no exemplo 7.9, é a função interna `averager`. De qualquer maneira, simplesmente chamamos `avg(n)` para incluir `n` na série e obter a média atualizada.

O local em que `avg` da classe `Averager` mantém o histórico é óbvio: no atributo de instância `self.series`. Mas em que lugar a função `avg` no segundo exemplo encontra `series`?

Observe que `series` é uma variável local de `make_averager` porque a inicialização `series = []` está no corpo dessa função. Porém, quando `avg(10)` é chamado, `make_averager` já retornou e seu escopo local não existe mais.

Em `averager`, `series` é uma *variável livre* (*free variable*). Esse é um termo técnico que indica que uma variável não tem uma associação no escopo local. Veja a figura 7.1.

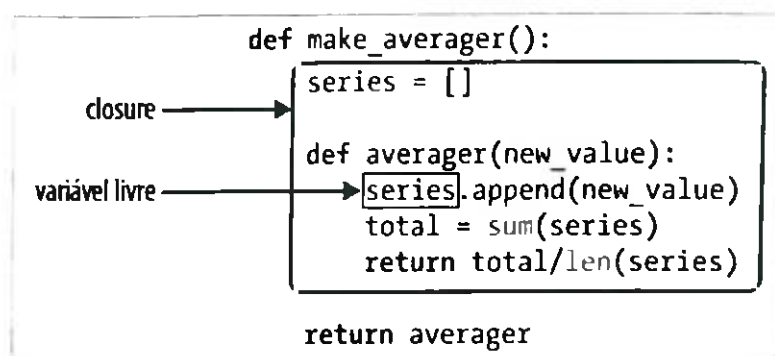


Figura 7.1 – A closure de `averager` estende o escopo dessa função para incluir a associação (binding) da variável livre `series`.

A inspeção do objeto `averager` devolvido mostra como Python mantém os nomes das variáveis locais e livres no atributo `__code__`, que representa o corpo compilado da função. O exemplo 7.11 mostra isso.

Exemplo 7.11 – Inspeccionando a função criada por `make_averager` no exemplo 7.9.

```
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

A associação com `series` é mantida no atributo `__closure__` da função `avg` devolvida. Cada item em `avg.__closure__` corresponde a um nome em `avg.__code__.co_freevars`. Esses itens são `cells` e têm um atributo chamado `cell_contents` em que o valor propriamente dito pode ser encontrado. O exemplo 7.12 mostra esses atributos.

Exemplo 7.12 – Continuação do exemplo 7.10

```
>>> avg.__code__.co_freevars
('series',)
>>> avg.__closure__
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

Para resumir: uma closure é uma função que preserva as associações com as variáveis livres existentes quando a função é definida, de modo que elas possam ser usadas posteriormente quando a função for chamada e o escopo de definição não estiver mais disponível.

Note que a única situação em que uma função pode precisar lidar com variáveis externas que não sejam globais é quando ela está aninhada em outra função.

Declaração nonlocal

Nossa implementação anterior de `make_averager` não é eficiente. No exemplo 7.9, armazenamos todos os valores da série histórica e calculamos a soma com `sum` sempre que `averager` era chamada. Uma implementação melhor seria simplesmente armazenar o total e a quantidade de itens até o momento e calcular a média a partir desses dois números.

O exemplo 7.13 é uma implementação com falha, somente para enfatizar uma questão. Você é capaz de ver em que ponto a falha ocorre?

Exemplo 7.13 – Uma função de ordem superior com falha para tentar calcular uma média em evolução sem manter todo o histórico

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

Se testar o exemplo 7.13, eis o que você verá:

```
>>> avg = make_averager()
>>> avg(10)
Traceback (most recent call last):
...
UnboundLocalError: local variable 'count' referenced before assignment
>>>
```

O problema é que o comando `count += 1` faz o mesmo que `count = count + 1` quando `count` é um número ou qualquer tipo imutável. Então, na verdade, estamos fazendo uma atribuição a `count` no corpo de `averager`, e isso faz dela uma variável local. O mesmo problema afeta a variável `total`.

Não tínhamos esse problema no exemplo 7.9 porque não fizemos nenhuma atribuição ao nome `series`; somente usamos `series.append` e chamamos `sum` e `len`. Desse modo, aproveitamos o fato de as listas serem mutáveis.

Porém, com tipos imutáveis como números, strings, tuplas etc., tudo que você pode fazer é ler, mas jamais atualizar. Se tentar refazer uma associação, como em `count = count + 1`, você estará implicitamente criando uma variável local `count`. Ela deixará de ser uma variável livre e, sendo assim, não será salva na closure.

Para contornar esse problema, a declaração `nonlocal` foi introduzida em Python 3. Ela permite sinalizar uma variável como livre, mesmo que ela receba um novo valor na função. Se um novo valor for atribuído a uma variável `nonlocal`, a associação armazenada na closure será alterada. Uma implementação correta de nosso mais novo `make_averager` está no exemplo 7.14.

Exemplo 7.14 – Calcula uma média em evolução sem manter todo o histórico (fixo com o uso de `nonlocal`)

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```



Sobrevivendo sem `nonlocal` em Python 2

A falta de `nonlocal` em Python 2 exige soluções alternativas, uma das quais está descrita no terceiro trecho de código da *PEP 3104* — *Access to Names in Outer Scopes* (Acesso a nomes em escopos externos, <http://www.python.org/dev/peps/pep-3104/>), que introduziu `nonlocal`. Essencialmente, a ideia é armazenar as variáveis que as funções internas precisam alterar (por exemplo, `count`, `total`) como itens ou atributos de algum objeto mutável, como um `dict` ou uma instância simples, e associar esse objeto a uma variável livre.

Agora que já discutimos as closures de Python, podemos implementar os decoradores de modo eficiente com funções aninhadas.

Implementando um decorador simples

O exemplo 7.15 mostra um decorador que calcula a duração de toda chamada à função decorada e exibe o tempo decorrido, os argumentos passados e o resultado da chamada.

Exemplo 7.15 – Um decorador simples para apresentar o tempo de execução das funções

```
import time

def clock(func):
    def clocked(*args): # ❶
        t0 = time.perf_counter()
        result = func(*args) # ❷
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked # ❸
```

- ❶ Define a função interna `clocked` para que aceite qualquer quantidade de argumentos posicionais.
- ❷ Essa linha funciona somente porque a closure para `clocked` inclui a variável livre `func`.
- ❸ Devolve a função interna para substituir a função decorada.

O exemplo 7.16 mostra o uso do decorador `clock`.

Exemplo 7.16 – Usando o decorador `clock`

```
# clockdeco_demo.py

import time
from clockdeco import clock

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

if __name__ == '__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(6)')
    print('6! =', factorial(6))
```

A saída da execução do exemplo 7.16 fica assim:

```
$ python3 clockdeco_demo.py
***** Calling snooze(123)
[0.12405610s] snooze(.123) -> None
***** Calling factorial(6)
[0.00000191s] factorial(1) -> 1
[0.00004911s] factorial(2) -> 2
[0.00008488s] factorial(3) -> 6
[0.00013208s] factorial(4) -> 24
[0.00019193s] factorial(5) -> 120
[0.00026107s] factorial(6) -> 720
6! = 720
```

Funcionamento

Lembre-se de que este código:

```
@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

na verdade, faz isto:

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

factorial = clock(factorial)
```

Sendo assim, em ambos os exemplos, `clock` recebe a função `factorial` como seu argumento `func` (veja o exemplo 7.15). Ela então cria e devolve a função `clocked`, que o interpretador Python atribui a `factorial` quando não estamos olhando. De fato, se você importar o módulo `clockdeco_demo` e consultar `__name__` de `factorial`, você obterá o seguinte:

```
>>> import clockdeco_demo
>>> clockdeco_demo.factorial.__name__
'clocked'
>>>
```

Portanto `factorial` agora armazena uma referência à função `clocked`. A partir de agora, sempre que `factorial(n)` for chamada, `clocked(n)` será executada. Essencialmente, `clocked` faz o seguinte:

1. Registra o tempo inicial `t0`.
2. Chama a função `factorial` original, salvando o resultado.
3. Calcula o tempo decorrido.

4. Formata e exibe os dados reunidos.
5. Devolve o resultado salvo no passo 2.

Esse é o comportamento típico de um decorador: ele substitui a função decorada por uma nova função que aceita os mesmos argumentos e (normalmente) devolve o que a função decorada deveria devolver, além de fazer outros processamentos também.



No livro *Design Patterns* de Gamma et al., a breve descrição do padrão Decorator (Decorador) começa assim: “Confere responsabilidades adicionais a um objeto dinamicamente.” Os decoradores de função se enquadram nessa descrição. Contudo, em nível de implementação, os decoradores Python apresentam pouca semelhança com o Decorator clássico descrito na obra *Design Patterns* original. A seção “Ponto de vista” na página 256 tem mais informações sobre esse assunto.

O decorador `clock` implementado no exemplo 7.15 tem algumas deficiências: ele não aceita argumentos nomeados e mascara `__name__` e `__doc__` da função decorada. O exemplo 7.17 usa o decorador `functools.wraps` para copiar os atributos relevantes de `func` para `clocked`. Além disso, nessa nova versão, os argumentos nomeados são tratados corretamente.

Exemplo 7.17 – Um decorador `clock` melhorado

```
# clockdeco2.py

import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - t0
        name = func.__name__
        arg_lst = []
        if args:
            arg_lst.append(', '.join(repr(arg) for arg in args))
        if kwargs:
            pairs = ['%s=%r' % (k, w) for k, w in sorted(kwargs.items())]
            arg_lst.append(', '.join(pairs))
        arg_str = ', '.join(arg_lst)
        print('[%0.8fs] %s(%s) -> %r ' % (elapsed, name, arg_str, result))
        return result
    return clocked
```

`functools.wraps` é somente um dos decoradores prontos para uso da biblioteca-padrão. Na próxima seção, conheceremos dois dos decoradores mais impressionantes oferecidos por `functools`: `lru_cache` e `singledispatch`.

Decoradores da biblioteca-padrão

Python tem três funções embutidas criadas para decorar métodos: `property`, `classmethod` e `staticmethod`. Discutiremos `property` na seção “Usando uma propriedade para validação de atributo” na página 665 e os demais em “`classmethod` *versus* `staticmethod`” na página 293.

Outro decorador visto com frequência é `functools.wraps`, um auxiliar para criar decoradores bem comportados. Nós o usamos no exemplo 7.17. Dois dos decoradores mais interessantes da biblioteca-padrão são `lru_cache` e o novíssimo `singledispatch` (acrescentado em Python 3.4). Ambos estão definidos no módulo `functools` e serão discutidos a seguir.

Memoização com `functools.lru_cache`

Um decorador bem prático é `functools.lru_cache`. Ele implementa memoização: uma técnica de otimização que funciona salvando os resultados de chamadas prévias a uma função custosa, evitando repetir processamentos em argumentos usados anteriormente. As letras LRU querem dizer Least Recently Used (usado menos recentemente), que quer dizer que o crescimento da cache é limitado, descartando-se as entradas que não tiverem sido lidas recentemente.

Uma boa demonstração é aplicar `lru_cache` à função recursiva extremamente lenta que gera o enésimo número da sequência de Fibonacci, como mostra o exemplo 7.18.

Exemplo 7.18 – A maneira recursiva bastante custosa de calcular o enésimo número da série de Fibonacci

```
from clockdeco import clock

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__ == '__main__':
    print(fibonacci(6))
```

Eis o resultado da execução de `fibonacci_demo.py`. Exceto pela última linha, toda a saída é gerada pelo decorador `clock`:

```
$ python3 fibo_demo.py
[0.000000095s] fibonacci(0) -> 0
[0.000000095s] fibonacci(1) -> 1
[0.00007892s] fibonacci(2) -> 1
[0.000000095s] fibonacci(1) -> 1
[0.000000095s] fibonacci(0) -> 0
[0.000000095s] fibonacci(1) -> 1
[0.00003815s] fibonacci(2) -> 1
[0.00007391s] fibonacci(3) -> 2
[0.00018883s] fibonacci(4) -> 3
[0.000000000s] fibonacci(1) -> 1
[0.000000095s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00004911s] fibonacci(2) -> 1
[0.00009704s] fibonacci(3) -> 2
[0.000000000s] fibonacci(0) -> 0
[0.000000000s] fibonacci(1) -> 1
[0.00002694s] fibonacci(2) -> 1
[0.000000095s] fibonacci(1) -> 1
[0.000000095s] fibonacci(0) -> 0
[0.000000095s] fibonacci(1) -> 1
[0.00005102s] fibonacci(2) -> 1
[0.00008917s] fibonacci(3) -> 2
[0.00015593s] fibonacci(4) -> 3
[0.00029993s] fibonacci(5) -> 5
[0.00052810s] fibonacci(6) -> 8
8
```

O desperdício é evidente: `fibonacci(1)` é chamado oito vezes, `fibonacci(2)` é chamado cinco vezes etc. Entretanto, se simplesmente adicionarmos duas linhas para usar `lru_cache`, o desempenho melhorará muito. Veja o exemplo 7.19.

Exemplo 7.19 – Implementação mais rápida usando caching

```
import functools
from clockdeco import clock
@functools.lru_cache() # ❶
@clock # ❷
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)
```

```
if __name__ == '__main__':
    print(fibonacci(6))
```

- ❶ Note que `lru_cache` deve ser chamada como um função normal – observe os parênteses na linha: `@functools.lru_cache()`. Isso se deve ao fato de ela aceitar parâmetros de configuração, como veremos em breve.
- ❷ Esse é um exemplo de decoradores empilhados: `@lru_cache()` é aplicado à função devolvida por `@clock`.

O tempo de execução cai para a metade e a função é chamada somente uma vez para cada valor de `n`:

```
$ python3 fibo_demo_lru.py
[0.00000119s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00010800s] fibonacci(2) -> 1
[0.00000787s] fibonacci(3) -> 2
[0.00016093s] fibonacci(4) -> 3
[0.00001216s] fibonacci(5) -> 5
[0.00025296s] fibonacci(6) -> 8
```

Em outro teste, para calcular `fibonacci(30)`, o exemplo 7.19 fez as 31 chamadas necessárias em 0,0005s, enquanto o exemplo 7.18 sem cache chamou `fibonacci` 2.692.537 vezes e demorou 17,7 segundos em um notebook Intel Core i7.

Além de viabilizar o uso de algoritmos recursivos tolos, `lru_cache` realmente se destaca em aplicações que precisam buscar informações na Web.

É importante observar que `lru_cache` pode ser ajustado por meio de dois argumentos opcionais. Sua assinatura completa é:

```
functools.lru_cache(maxsize=128, typed=False)
```

O argumento `maxsize` determina quantos resultados de chamadas são armazenados. Depois que o cache estiver cheio, os resultados mais antigos serão descartados para criar espaço. Para um desempenho otimizado, `maxsize` deve ser uma potência de 2. O argumento `typed`, se estiver definido com `True`, fará os resultados de argumentos de tipos diferentes serem armazenados separadamente, ou seja, diferenciará argumentos de ponto flutuante de inteiros que, normalmente, são considerados iguais, como 1 e 1.0. A propósito, como `lru_cache` usa um `dict` para armazenar os resultados e as chaves são compostas a partir dos argumentos posicionais e nomeados usados nas chamadas, todos os argumentos recebidos pela função decorada devem ser *hashable*.

Vamos agora considerar o curioso decorador `functools singledispatch`.

Funções genéricas com dispatch simples

Suponha que estejamos criando uma ferramenta para depurar aplicações web. Queremos gerar visualizações em HTML para diferentes tipos de objetos Python.

Poderíamos começar com uma função como:

```
import html

def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}/pre>'.format(content)
```

Esse código funciona para qualquer tipo Python, mas agora queremos estendê-lo para que gere visualizações personalizadas para alguns tipos:

- `str`: substitui caracteres de quebra de linha por '`
\n`' e usa tags `<p>` em vez de `<pre>`.
- `int`: mostra o número em decimal e em hexadecimal.
- `list`: gera uma lista HTML, formatando cada item de acordo com seu tipo.

O comportamento que queremos está no exemplo 7.20.

Exemplo 7.20 – `htmlize` gera HTML personalizado conforme os diferentes tipos de objeto

```
>>> htmlize({1, 2, 3}) ❶
'<pre>{1, 2, 3}</pre>'
>>> htmlize(abs)
'<pre>&lt;built-in function abs&gt;</pre>'
>>> htmlize('Heimlich & Co.\n- a game') ❷
'<p>Heimlich & Co.<br>\n- a game</p>'
>>> htmlize(42) ❸
'<pre>42 (0x2a)</pre>'
>>> print(htmlize(['alpha', 66, {3, 2, 1}])) ❹
<ul>
<li><p>alpha</p></li>
<li><pre>66 (0x42)</pre></li>
<li><pre>{1, 2, 3}</pre></li>
</ul>
```

- ❶ Por padrão, a `repr` de um objeto com escape HTML é mostrada entre `<pre></pre>`.
- ❷ Objetos `str` também têm escape HTML, mas são colocados entre `<p></p>` com quebras de linha `
`.
- ❸ Um `int` é mostrado em decimal e em hexadecimal entre `<pre></pre>`.

- ❶ Cada item da lista é formatado segundo o seu tipo e a sequência toda é apresentada como uma lista HTML.

Como não temos sobrecarga de método nem de função em Python, não podemos criar variações de `htmlize` com assinaturas diferentes para cada tipo de dado que queremos tratar de modo diferente. Uma solução comum em Python seria transformar `htmlize` em uma função de despacho (*dispatch*), com uma cadeia de `if/elif/elif` chamando funções especializadas como `htmlize_str`, `htmlize_int` etc. Essa solução não é extensível pelos usuários de nosso módulo, além de não ser prática: com o tempo, o dispatcher `htmlize` ficaria grande demais e o acoplamento entre ele e as funções especializadas seria muito alto.

O novo decorador `functools singledispatch` em Python 3.4 permite que cada módulo contribua com a solução geral e deixa você fornecer facilmente uma função especializada, mesmo para classes que não possam ser alteradas. Se você decorar uma função simples com `@singledispatch`, ela se tornará uma *função genérica*: um grupo de funções para realizar a mesma operação de maneiras diferentes de acordo com o tipo do primeiro argumento.³ O exemplo 7.21 mostra como fazer isso.



`functools.singledispatch` foi adicionado em Python 3.4, mas o pacote `singledispatch` (<https://pypi.python.org/pypi/singledispatch>) disponível em PyPI funciona com Python de 2.6 a 3.3.

Exemplo 7.21 – `singledispatch` cria um `htmlize.register` personalizado para reunir várias funções em uma função genérica

```
from functools import singledispatch
from collections import abc
import numbers
import html

@singledispatch ❶
def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}/pre>'.format(content)

@htmlize.register(str) ❷
def _(text): ❸
    content = html.escape(text).replace('\n', '<br>\n')
    return '<p>{}/p>'.format(content)
```

³ É isso que o termo despacho simples (single-dispatch) quer dizer. Se mais argumentos fossem usados para seleccionar as funções específicas, teríamos despacho múltiplo (multiple-dispatch).

```
@htmlize.register(numbers.Integral) ❹
def _(n):
    return '<pre>{0} (0x{0:x})</pre>'.format(n)

@htmlize.register(tuple) ❺
@htmlize.register(abc.MutableSequence)
def _(seq):
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'
```

- ❶ `@singledispatch` marca a função-base que trata o tipo `object`.
- ❷ Cada função especializada é decorada com `@«função-base».register(«tipo»)`.
- ❸ O nome das funções especializadas é irrelevante; `_` é uma boa opção para deixar isso claro.
- ❹ Para cada tipo adicional que vá receber um tratamento especial, registre uma nova função. `numbers.Integral` é uma superclasse virtual de `int`.
- ❺ Você pode empilhar diversos decoradores `register` para dar suporte a tipos diferentes com a mesma função.

Quando possível, registre as funções especializadas para que tratem ABCs (classes abstratas), como `numbers.Integral` e `abc.MutableSequence`, em vez de implementações concretas como `int` e `list`. Isso permite que seu código trate uma variedade maior de tipos compatíveis. Por exemplo, uma extensão Python pode oferecer alternativas ao tipo `int` com número fixo de bits como subclasses de `numbers.Integral`.



O uso de ABCs para verificação de tipo permite que seu código dê suporte a classes existentes ou futuras que sejam realmente subclasses ou subclasses virtuais dessas ABCs. O uso de ABCs e o conceito de subclasse virtual serão discutidos no capítulo 11.

Uma característica digna de nota do funcionamento de `singledispatch` é que você pode registrar funções especializadas em qualquer lugar do sistema, em qualquer módulo. Mais tarde, se adicionar um módulo com um novo tipo definido pelo usuário, poderá facilmente oferecer uma nova função personalizada para tratar esse tipo. Além disso, poderá escrever funções personalizadas para classes que não tenha escrito ou que não possa alterar.

O decorador `singledispatch` é um acréscimo muito bem pensado à biblioteca-padrão e oferece mais recursos do que podemos descrever aqui. A melhor documentação para ele é a *PEP 443 – Single-dispatch generic functions* (Funções genéricas de despacho simples, <https://www.python.org/dev/peps/pep-0443/>).



`@singledispatch` não foi criado para trazer sobrecarga de métodos em estilo Java para Python. Uma única classe com muitas variantes para sobrecarga de um método é melhor que uma única função com um trecho longo de blocos `if/elif/elif/elif`. Porém as duas soluções são imperfeitas, pois concentram muita responsabilidade em uma única unidade de código – a classe ou a função. A vantagem de `@singledispatch` é oferecer suporte à extensão modular: cada módulo pode registrar uma função especializada para cada tipo suportado por ele.

Decoradores são funções e, sendo assim, podem ser compostos (ou seja, você pode aplicar um decorador em uma função que já esteja decorada, como mostra o exemplo 7.21). A seção a seguir explica como isso funciona.

Decoradores empilhados

O exemplo 7.19 mostrou o uso de decoradores empilhados: `@lru_cache` é aplicado no resultado de `@clock` em `fibonacci`. No exemplo 7.21, o decorador `@htmlize.register` foi aplicado duas vezes na última função do módulo.

Quando dois decoradores `@d1` e `@d2` são aplicados a uma função `f` nessa ordem, o resultado será o mesmo que `f = d1(d2(f))`.

Em outras palavras, isto:

```
@d1
@d2
def f():
    print('f')
```

é o mesmo que:

```
def f():
    print('f')

f = d1(d2(f))
```

Além de decoradores empilhados, este capítulo mostrou alguns decoradores que aceitam argumentos, por exemplo, `@lru_cache()` e `htmlize.register(«tipo»)` produzido por `@singledispatch` no exemplo 7.21. A próxima seção mostra como criar decoradores que aceitam parâmetros.

Decoradores parametrizados

Ao fazer parse de um decorador no código-fonte, Python pega a função decorada e passa-a como o primeiro argumento para a função decoradora. Então como você faz um decorador aceitar outros argumentos? A resposta é: crie uma fábrica (factory) de decoradores que aceite esses argumentos e devolva um decorador que, por sua vez, será aplicado à função a ser decorada. Confuso? Certamente. Vamos começar com um exemplo baseado no decorador mais simples que vimos: `register` no exemplo 7.22.

Exemplo 7.22 – Módulo `registration.py` do exemplo 7.2 condensado, repetido aqui por conveniência

```
registry = []

def register(func):
    print('running register(%s)' % func)
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')

print('running main()')
print('registry ->', registry)
f1()
```

Um decorador de registro parametrizado

Para tornar mais fácil habilitar ou desabilitar o registro de função por `register`, faremos ele aceitar um parâmetro `active` opcional que, se for `False`, não registrará a função decorada. O exemplo 7.23 mostra como fazer isso. Conceitualmente, a nova função `register` não é um decorador, mas uma fábrica de decoradores. Quando chamada, essa função devolve o verdadeiro decorador que será aplicado à função-alvo.

Exemplo 7.23 – Para aceitar parâmetros, o novo decorador `register` deve ser chamado como uma função

```
registry = set() ❶

def register(active=True): ❷
    def decorate(func): ❸
        print('running register(active=%s)->decorate(%s)'
              % (active, func))
        if active: ❹
            registry.add(func)
        else:
```

```

        registry.discard(func) ❸

    return func ❹
    return decorate ❺

@register(active=False) ❻
def f1():
    print('running f1()')

@register() ❼
def f2():
    print('running f2()')

def f3():
    print('running f3()')

```

- ❶ registry agora é um set, portanto adicionar e remover funções é mais rápido.
- ❷ register aceita um argumento nomeado opcional.
- ❸ A função interna decorate é o verdadeiro decorador; observe como ela aceita uma função como argumento.
- ❹ Registra func somente se o argumento active (recuperado da closure) for True.
- ❺ Se not active e func in registry, remove a função.
- ❻ Como decorate é um decorador, ele precisa devolver uma função.
- ❼ register é nossa fábrica de decoradores, portanto ela devolve decorate.
- ❽ A fábrica @register deve ser chamada como uma função, com os parâmetros desejados.
- ❾ Se nenhum parâmetro for passado, register ainda deve ser chamado como uma função – @register() – para devolver o verdadeiro decorador decorate.

O ponto principal é que register() devolve decorate, que, por sua vez, será aplicado à função decorada.

O código do exemplo 7.23 está em um módulo *registration_param.py*. Se importarmos esse módulo, veremos o seguinte:

```

>>> import registration_param
running register(active=False)->decorate(<function f1 at 0x10063c1e0>)
running register(active=True)->decorate(<function f2 at 0x10063c268>)
>>> registration_param.registry
[<function f2 at 0x10063c268>]

```

Observe como somente a função f2 aparece em registry; f1 não aparece porque active=False foi passado à fábrica de decoradores register, portanto o decorate aplicado a f1 não o adicionou a registry.

Se, em vez de usar a sintaxe @, tivéssemos usado `register` como uma função normal, a sintaxe necessária para decorar uma função `f` seria `register()(f)` para adicionar `f` ao `registry`, ou `register(active=False)(f)` para não adicioná-la (ou removê-la). Veja o exemplo 7.24, que tem uma demo para adicionar e remover funções de `registry`.

Exemplo 7.24 – Usando o módulo `registration_param` exibido no exemplo 7.23

```
>>> from registration_param import *
running register(active=False)->decorate(<function f1 at 0x10073c1e0>)
running register(active=True)->decorate(<function f2 at 0x10073c268>)
>>> registry # ❶
{<function f2 at 0x10073c268>}
>>> register()(f3) # ❷
running register(active=True)->decorate(<function f3 at 0x10073c158>)
<function f3 at 0x10073c158>
>>> registry # ❸
{<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
>>> register(active=False)(f2) # ❹
running register(active=False)->decorate(<function f2 at 0x10073c268>)
<function f2 at 0x10073c268>
>>> registry # ❺
{<function f3 at 0x10073c158>}
```

- ❶ Quando o módulo é importado, `f2` está em `registry`.
- ❷ A expressão `register()` devolve `decorate`, que então é aplicado a `f3`.
- ❸ A linha anterior adicionou `f3` a `registry`.
- ❹ Essa chamada remove `f2` de `registry`.
- ❺ Confirma que apenas `f3` permaneceu em `registry`.

O funcionamento dos decoradores parametrizados é complicado, e o que acabamos de discutir é mais simples que a maioria. Os decoradores parametrizados normalmente substituem a função decorada, e sua construção exige mais um nível de aninhamento. Explorar por essas pirâmides de funções é nossa próxima aventura.

Decorador `clock` parametrizado

Nesta seção, retomamos o decorador `clock`, acrescentando um recurso: os usuários poderão passar uma string de formatação para controlar a saída da função decorada. Veja o exemplo 7.25.



Por questões de simplicidade, o exemplo 7.25 é baseado na implementação inicial de `clock` do exemplo 7.15, e não na versão melhorada do exemplo 7.17 que usa `@functools.wraps`, acrescentando mais uma camada de função.

Exemplo 7.25 – Módulo `clockdeco_param.py`: o decorador `clock` parametrizado

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT): ❶
    def decorate(func):      ❷
        def clocked(*_args): ❸
            t0 = time.time()
            _result = func(*_args) ❹
            elapsed = time.time() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args) ❺
            result = repr(_result) ❻
            print(fmt.format(**locals())) ❼
            return _result ❽
        return clocked ❾
    return decorate ❿

if __name__ == '__main__':
    @clock() ⓫
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(.123)
```

- ❶ `clock` é nossa fábrica de decoradores parametrizada.
- ❷ `decorate` é o verdadeiro decorador.
- ❸ `clocked` encapsula a função decorada.
- ❹ `_result` é o verdadeiro resultado da função decorada.
- ❺ `_args` armazena os argumento de `clocked`, enquanto `args` é o str usado para exibição.
- ❻ `result` é a representação em str de `_result` para exibição.
- ❼ Usar `**locals()` nesse caso permite que qualquer variável local de `clocked` seja referenciada em `fmt`.

- ⑧ `clocked` substituirá a função decorada, portanto deve devolver o que essa função devolve.
- ⑨ `decorate` devolve `clocked`.
- ⑩ `clock` devolve `decorate`.
- ⑪ Nesse teste, `clock()` é chamado sem argumentos, portanto o decorador aplicado usará o `str` de formatação default.

Se executar o exemplo 7.25 do shell, você verá o seguinte:

```
$ python3 clockdeco_param.py
[0.12412500s] snooze(0.123) -> None
[0.12411904s] snooze(0.123) -> None
[0.12410498s] snooze(0.123) -> None
```

Para exercitar a nova funcionalidade, os exemplos 7.26 e 7.27 mostram dois outros módulos que usam `clockdeco_param` e as saídas geradas.

Exemplo 7.26 – `clockdeco_param_demo1.py`

```
import time
from clockdeco_param import clock

@clock('{name}: {elapsed}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Saída do exemplo 7.26:

```
$ python3 clockdeco_param_demo1.py
snooze: 0.12414693832397461s
snooze: 0.1241159439086914s
snooze: 0.12412118911743164s
```

Exemplo 7.27 – `clockdeco_param_demo2.py`

```
import time
from clockdeco_param import clock

@clock('{name}({args}) dt={elapsed:0.3f}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Saída do exemplo 7.27:

```
$ python3 clockdeco_param_demo2.py
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
```

Com isso, encerramos nossa exploração dos decoradores no espaço permitido pelo escopo deste livro. Veja a seção “Leituras complementares” na próxima página, em particular, o blog de Graham Dumpleton e o módulo `wrapt` para ver técnicas extremamente robustas para a criação de decoradores.



Graham Dumpleton e Lennart Regebro – um dos revisores técnicos deste livro – argumentam que é melhor escrever os decoradores como classes que implementem `__call__`, e não como funções, como nos exemplos deste capítulo. Concordo que essa abordagem seja melhor para decoradores não triviais, mas, para explicar a ideia básica desse recurso da linguagem, as funções são mais fáceis de entender.

Resumo do capítulo

Avançamos bastante neste capítulo, mas tentei deixar a jornada tão suave quanto possível, mesmo quando o terreno era acidentado. Afinal de contas, entramos nos domínios da metaprogramação.

Começamos com um decorador `@register` simples, sem nenhuma função interna, e terminamos com um decorador `@clock()` parametrizado envolvendo dois níveis de funções aninhadas.

Os decoradores de registro, embora simples em sua essência, têm aplicações reais em frameworks Python sofisticados. Aplicamos a ideia de registro para melhorar nossa refatoração do padrão de projeto Strategy do capítulo 6.

Decoradores parametrizados quase sempre envolvem pelo menos duas funções aninhadas, talvez mais, se você quiser usar `@functools.wraps` para produzir um decorador que ofereça melhor suporte a técnicas mais sofisticadas. Uma dessas técnicas são os decoradores empilhados, que discutimos rapidamente.

Também vimos dois decoradores de função sensacionais incluídos no módulo `functools` da biblioteca-padrão: `@lru_cache()` e `@singledispatch`.

Entender como os decoradores realmente funcionam exigiu abordar a diferença entre *tempo de importação* e *tempo de execução* e, em seguida, explorar escopos de variáveis, closures e a nova declaração `nonlocal`. Dominar closures e `nonlocal` é importante não

só para criar decoradores, mas também para escrever programas orientados a eventos para GUIs ou I/O assíncrono com callbacks.

Leituras complementares

O capítulo 9 – “Metaprogramming” (Metaprogramação) – de *Python Cookbook*, 3ª edição, de David Beazley e Brian K. Jones (O’Reilly),⁴ tem várias receitas, de decoradores elementares a outros bem sofisticados, incluindo um que pode ser chamado como um decorador normal ou como uma fábrica de decoradores, por exemplo, `@clock` ou `@clock()`. É a “Recipe 9.6. Defining a Decorator That Takes an Optional Argument” (Definir um decorador que aceite um argumento opcional) do cookbook.

Graham Dumpleton tem uma série de posts de blog detalhados (<http://bit.ly/1DePPcl>) sobre técnicas para implementar decoradores bem comportados, começando com “How You Implemented Your Python Decorator is Wrong” (O modo como você implementava seu decorador Python está errado, <http://bit.ly/1DePVRi>). Seu profundo expertise nessa questão também aparece no módulo `wrapt` (<http://wrapt.readthedocs.org/en/latest/>), escrito por ele para simplificar a implementação de decoradores e wrappers dinâmicos de função que tratem introspecção e se comportem corretamente quando adicionalmente decorados, ao serem aplicados a métodos ou usados como descritores. (Os descritores são o assunto do capítulo 20.)

Michele Simionato é autor de um pacote cujo objetivo é “simplificar o uso de decoradores para o programador médio e popularizar os decoradores ao mostrar diversos exemplos não triviais”, de acordo com a documentação. Esse pacote está disponível em PyPI como o pacote `decorator` (<https://pypi.python.org/pypi/decorator>).

Criada quando os decoradores ainda eram um recurso novo em Python, a página wiki de *Python Decorator Library* (Biblioteca de decoradores Python, <https://wiki.python.org/moin/PythonDecoratorLibrary>) tem dezenas de exemplos. Como essa página foi iniciada há anos, algumas das técnicas mostradas se tornaram ultrapassadas, mas a página continua sendo uma excelente fonte de inspiração.

A PEP 443 (<http://www.python.org/dev/peps/pep-0443/>) oferece a justificativa e uma descrição detalhada para o recurso de funções genéricas de despacho simples. Um post de blog antigo (março de 2005) de Guido van Rossum, “Five-Minute Multimethods in Python” (Multimétodos em Python em cinco minutos, <http://www.artima.com/weblogs/viewpost.jsp?thread=101605>), descreve uma implementação de funções genéricas (também conhecidas como multimétodos) usando decoradores. Seu código oferece suporte para despacho múltiplo (ou seja, despacho baseado em mais de um argumento posicional). O código dos multimétodos de Guido é interessante, mas é um exemplo didático. Para uma im-

4 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

plementação moderna de funções genéricas com despacho múltiplo, pronta para o ambiente de produção, dê uma olhada em Reg (<http://reg.readthedocs.org/en/latest/>), de Martijn Faassen – autor do framework web Morepath (<http://morepath.readthedocs.org/en/latest/>) orientado a modelos e especializado em REST.

“Closures in Python” (<http://effbot.org/zone/closure.htm>) é um pequeno post de blog de Fredrik Lundh que explica a terminologia das closures.

A PEP 3104 – *Access to Names in Outer Scopes* (Acesso a nomes em escopos externos, <http://www.python.org/dev/peps/pep-3104/>) descreve a introdução da declaração `nonlocal` para permitir a reassociação de nomes que não sejam nem locais nem globais. Ela também inclui uma excelente visão geral de como esse problema é resolvido em outras linguagens dinâmicas (Perl, Ruby, JavaScript etc.) e os prós e contras das opções de design disponíveis em Python.

Em nível mais teórico, a PEP 227 – *Statically Nested Scopes* (Escopos estaticamente aninhados, <http://www.python.org/dev/peps/pep-0227/>) documenta a introdução de escopos léxicos como uma opção em Python 2.1 e como padrão em Python 2.2, explicando o raciocínio e as opções de design para a implementação de closures em Python.

Ponto de vista

O designer de qualquer linguagem com funções de primeira classe se depara com este problema: por serem objetos de primeira classe, as funções são definidas em um determinado escopo, mas podem ser chamadas em outros. A pergunta é: como avaliar as variáveis livres? A primeira resposta, e a mais simples, é usar “escopo dinâmico”. Isso quer dizer que as variáveis livres são avaliadas observando o ambiente em que a função é chamada.

Se Python tivesse escopos dinâmicos, mas não tivesse closures, poderíamos improvisar `avg` – semelhante ao exemplo 7.9 – desta maneira:

```
>>> ### esta não é uma sessão de console de Python de verdade! ###
>>> avg = make_averager()
>>> series = [] # ❶
>>> avg(10)
10.0
>>> avg(11) # ❷
10.5
>>> avg(12)
11.0
>>> series = [1] # ❸
>>> avg(5)
3.0
```

- ❶ Antes de usar `avg`, precisamos definir `series = []` por conta própria, portanto temos que saber que `averager` (em `make_averager`) refere-se a uma `list` com esse nome.
- ❷ Internamente, `series` é usado para reunir os valores cuja média será calculada.
- ❸ Quando `series = [1]` é executado, a lista anterior é perdida. Isso poderia acontecer acidentalmente, ao tratar duas médias independentes em evolução ao mesmo tempo.

Funções devem ser caixas-pretas, com implementações ocultas aos usuários. Entretanto, com escopos dinâmicos, se uma função usar variáveis livres, o programador deverá conhecer a operação interna para definir um ambiente em que isso funcione corretamente.

Por outro lado, o escopo dinâmico é mais fácil de implementar, e por isso, provavelmente, foi o caminho escolhido por John McCarthy quando criou Lisp, a primeira linguagem a ter funções de primeira classe. O artigo “The Roots of Lisp” (Raízes de Lisp, <http://www.paulgraham.com/rootsoflisp.html>) de Paul Graham tem uma explicação compreensível do artigo original de John McCarthy sobre a linguagem Lisp: “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I” (Funções recursivas de expressões simbólicas e seu processamento pelo computador, Parte I, http://bit.ly/mccarthy_recursive). O artigo de McCarthy é uma obra-prima tão grandiosa quanto a Nona Sinfonia de Beethoven. Paul Graham traduziu-o para nós, de matemática para inglês e para código executável.

O comentário de Paul Graham também mostra como o uso de escopos dinâmicos pode ser complicado. Esta é uma citação de “The Roots of Lisp”:

O fato de o primeiro exemplo de funções de ordem superior em Lisp estar quebrado por causa de escopos dinâmicos é um testemunho eloquente dos perigos que eles representam. Pode ser que McCarthy não estivesse totalmente ciente das implicações do escopo dinâmico em 1960. O escopo dinâmico permaneceu em implementações de Lisp durante um período de tempo surpreendentemente longo – até Sussman e Steele terem desenvolvido a linguagem Scheme em 1975. O escopo léxico não complica muito a definição de `eval`, mas pode dificultar a escrita de compiladores.

Atualmente, o escopo léxico é a norma: variáveis livres são avaliadas considerando o ambiente em que a função é definida. O escopo léxico complica a implementação de linguagens com funções de primeira classe, pois exige suporte a closures. Por outro lado, o escopo léxico deixa o código-fonte mais legível. A maioria das linguagens criadas desde Algol tem escopo léxico.

Durante muitos anos, os `lambdas` de Python não ofereciam closures, contribuindo para a péssima reputação desse recurso entre os geeks de programação funcional

na blogosfera. Isso foi corrigido em Python 2.2 (dezembro de 2001), mas a blogosfera tem uma memória de longa duração. Desde então, `lambda` deixa a desejar somente por causa de sua sintaxe limitada.

Decoradores em Python e o padrão de projeto Decorator

Os decoradores de função em Python se enquadram na descrição geral de Decorator dada por Gamma et al. no livro *Design Patterns*: “Confere responsabilidades adicionais a um objeto dinamicamente. Os Decorators oferecem uma alternativa flexível à criação de subclasses para estender funcionalidades.”

Em nível de implementação, os decoradores em Python não lembram o padrão de projeto clássico Decorator, mas uma analogia pode ser feita.

No padrão de projeto, Decorator e Component são classes abstratas. Uma instância de um decorador concreto engloba uma instância de um componente concreto para lhe acrescentar novos comportamentos. Fazendo uma citação de *Design Patterns*:

O decorador está de acordo com a interface do componente que ele decora, de modo que sua presença será transparente aos clientes do componente. O decorador encaminha solicitações ao componente e pode realizar ações adicionais (por exemplo, desenhar uma borda) antes ou depois do encaminhamento. A transparência permite que você aninhe decoradores recursivamente, permitindo assim um número ilimitado de responsabilidades adicionais. (p. 175)

Em Python, a função decoradora desempenha o papel de uma subclasse concreta de Decorator, e a função interna que ela devolve é uma instância de decorador. A função devolvida encapsula a função a ser decorada, que é análoga ao componente no padrão de projeto. A função devolvida é transparente porque está de acordo com a interface do componente, pois aceita os mesmos argumentos. Ela encaminha chamadas ao componente e pode realizar ações adicionais, seja antes ou depois disso. Aproveitando a citação anterior, podemos reutilizar a última frase: “A transparência permite que você aninhe decoradores recursivamente, permitindo assim um número ilimitado de comportamentos adicionais.” É isso que permite que os decoradores empilhados funcionem.

Observe que não estou sugerindo que os decoradores de função devam ser usados para implementar o padrão Decorator em programas Python. Embora isso possa ser feito em situações específicas, em geral, é melhor implementar o padrão Decorator com classes que representem o Decorator e os componentes que elas irão embrulhar.

PARTE IV

Práticas de orientação a objetos

CAPÍTULO 8

Referências a objetos, mutabilidade e reciclagem

'Você está triste', disse o Cavaleiro ansioso, 'deixe-me cantar uma canção para consolá-la [...]. O nome da canção é chamado "OLHOS DE HADOQUE".'

'Ah, esse é o nome da canção, certo?', disse Alice, tentando se mostrar interessada.

'Não, você não entendeu', disse o Cavaleiro, parecendo um pouco contrariado. 'É assim que o nome é CHAMADO. O verdadeiro nome É "O VELHO HOMEM VELHO".'
(adaptação do capítulo VIII. 'É minha própria invenção').

– Lewis Carroll

Alice através do espelho e o que ela encontrou lá

Alice e o Cavaleiro definem o tom do que veremos neste capítulo. O tema é a distinção entre objetos e seus nomes. Um nome não é o objeto; um nome é algo diferente.

Começaremos o capítulo apresentando uma metáfora para variáveis em Python: variáveis são rótulos, e não caixas. Se variáveis de referência não são novidade para você, a analogia ainda poderá vir a calhar se precisar explicar problemas de aliasing (apelidamento) a outras pessoas.

Em seguida, discutiremos os conceitos de identidade de objeto, valor e apelidamento (quando um objeto tem mais de um nome). Um traço surpreendente das tuplas será revelado: elas são imutáveis, mas seus valores podem mudar. Isso leva a uma discussão sobre cópias rasas (shallow copy) e cópias profundas (deep copy). Referências e parâmetros de função serão o nosso próximo tema: o problema com defaults mutáveis para parâmetros e o tratamento seguro de argumentos mutáveis passados por clientes de nossas funções.

As últimas seções do capítulo abordam a coleta de lixo (garbage collection), o comando `del` e o uso de referências fracas (weak references) para "lembrar" objetos sem mantê-los vivos.

É um capítulo bem árido, mas seus assuntos estão no cerne de muitos bugs sutis em programas Python reais.

Vamos começar desaprendendo que uma variável é uma caixa em que você armazena dados.

Variáveis não são caixas

Em 1997, fiz um curso de verão sobre Java no MIT. Lynn Andrea Stein, a professora – uma educadora premiada da área de ciência da computação, que hoje dá aulas no Olin College of Engineering – argumentou que a metáfora usual “variáveis são caixas”, na verdade, atrapalha o entendimento de variáveis de referência em linguagens orientadas a objetos. As variáveis em Python são como variáveis de referência em Java, portanto é melhor pensar nelas como rótulos associados a objetos.

O exemplo 8.1 mostra uma interação simples, que não pode ser explicada pela ideia de “variáveis como caixas”. A figura 8.1 ilustra por que a metáfora de caixas é errada em Python, enquanto as etiquetas oferecem uma imagem conveniente de como as variáveis realmente funcionam.

Exemplo 8.1 – Variáveis *a* e *b* armazenam referências à mesma lista em vez de cópias

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

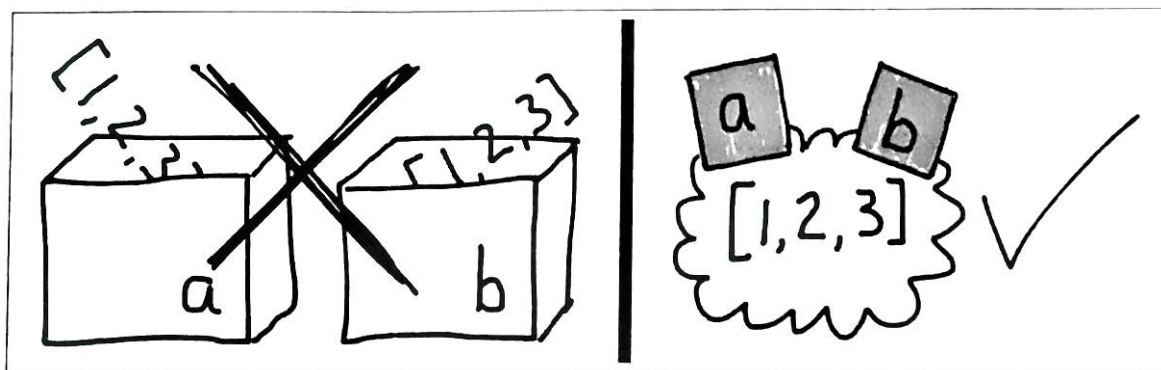


Figura 8.1 – Se imaginar que variáveis são como caixas, não há como entender a atribuição em Python; em vez disso, pense em variáveis como etiquetas – o exemplo 8.1 então será fácil de explicar.

A professora Stein também falava de atribuição de forma bastante deliberada. Por exemplo, ao falar sobre um objeto gangorra em uma simulação, ela dizia: “a variável *s* é atribuída à gangorra”, mas jamais “a gangorra é atribuída à variável *s*”. Com variáveis de referência, faz muito mais sentido dizer que a variável é atribuída a um objeto, e

não o contrário. Afinal de contas, o objeto é criado antes da atribuição. O exemplo 8.2 prova que o lado direito de uma atribuição ocorre antes.

Exemplo 8.2 – Variáveis são atribuídas a objetos somente depois que os objetos são criados

```
>>> class Gizmo:
...     def __init__(self):
...         print('Gizmo id: %d' % id(self))
...
>>> x = Gizmo()
Gizmo id: 4301489152 ❶
>>> y = Gizmo() * 10 ❷
Gizmo id: 4301489432 ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
>>>
>>> dir() ❹
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'x']
```

- ❶ A saída `Gizmo id: ...` é um efeito colateral da criação de uma instância de `Gizmo`.
- ❷ Multiplicar uma instância de `Gizmo` levantará uma exceção.
- ❸ Eis a prova de que um segundo `Gizmo` realmente foi instanciado antes da tentativa de multiplicação.
- ❹ Mas a variável `y` não foi criada, pois a exceção ocorreu enquanto o lado direito da atribuição estava sendo avaliado.



Para entender uma atribuição em Python, sempre leia o lado direito antes: é aí que o objeto é criado ou acessado. Depois disso, a variável à esquerda é associada ao objeto, como um rótulo grudado nele. Esqueça as caixas.

Como as variáveis são apenas rótulos, nada impede que um objeto tenha vários rótulos atribuídos a ele. Quando isso acontecer, você terá *apelidos* (aliases), que será o nosso próximo assunto.

Identidade, igualdade e apelidos

Lewis Carroll é o pseudônimo do professor Charles Lutwidge Dodgson. O Sr. Carroll não é apenas igual ao professor Dodgson: trata-se de uma só pessoa. O exemplo 8.3 expressa essa ideia em Python.

Exemplo 8.3 – charles e lewis se referem ao mesmo objeto

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles ❶
>>> lewis is charles
True
>>> id(charles), id(lewis) ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950 ❸
>>> charles
{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832}
```

❶ lewis é um apelido para charles.

❷ O operador `is` e a função `id` confirmam isso.

❸ Adicionar um item em lewis é o mesmo que adicionar um item em charles.

No entanto suponha que um impostor – vamos chamá-lo de Dr. Alexander Pedachenko – argumente que ele seja Charles L. Dodgson, nascido em 1832. Suas credenciais podem ser as mesmas, mas o Dr. Pedachenko não é o professor Dodgson. A figura 8.2 ilustra esse cenário.

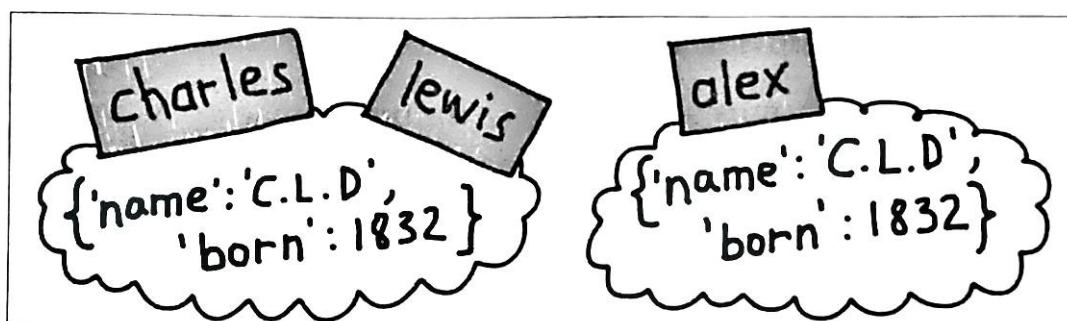


Figura 8.2 – charles e lewis estão associados ao mesmo objeto; alex está associado a um objeto diferente de conteúdo igual.

O exemplo 8.4 implementa e testa o objeto alex representado na figura 8.2.

Exemplo 8.4 – alex e charles são comparados como iguais, mas alex não é charles

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950} ❶
>>> alex == charles ❷
True
>>> alex is not charles ❸
True
```

❶ alex refere-se a um objeto que é uma réplica do objeto atribuído a charles.

❷ Os objetos são comparados como iguais por causa da implementação de `__eq__` na classe `dict`.

- ❸ Porém são objetos distintos. Essa é a maneira pythônica de escrever a negação da comparação de identidade: `a is not b`.

O exemplo 8.3 é um exemplo de *apelidamento* (aliasing). Nesse código, `lewis` e `charles` são apelidos (aliases): duas variáveis associadas ao mesmo objeto. Por outro lado, `alex` não é um apelido de `charles`: essas variáveis estão associadas a objetos distintos. Os objetos associados a `alex` e a `charles` têm o mesmo *valor* – é isso que `==` compara – mas têm identidades diferentes.

Em *The Python Language Reference* (Guia de referência à linguagem Python), a seção “3.1. Objects, values and types” (Objetos, valores e tipos, <http://bit.ly/1Vm9gv4>) afirma o seguinte:

Todo objeto tem uma identidade, um tipo e um valor. A identidade de um objeto não muda depois que esse objeto é criado; você pode pensar nela como o endereço do objeto na memória. O operador `is` compara a identidade de dois objetos; a função `id()` devolve um inteiro que representa a identidade do objeto.

O verdadeiro significado do ID de um objeto depende da implementação. Em CPython, `id()` devolve o endereço de memória do objeto, mas pode ser um valor diferente em outro interpretador Python. O ponto principal é que o interpretador garante que o ID seja um rótulo numérico único e que jamais sofrerá alterações durante o tempo de vida do objeto.

Na prática, raramente usamos a função `id()` na programação. As verificações de identidade são mais frequentes com o operador `is`, e não pela comparação de IDs. A seguir, falaremos sobre `is` *versus* `==`.

Escolhendo entre `==` e `is`

O operador `==` compara valores de objetos (os dados que eles armazenam), enquanto `is` compara suas identidades.

Geralmente estamos interessados nos valores, e não nas identidades, portanto `==` aparece com mais frequência que `is` em códigos Python.

No entanto, ao comparar uma variável com um singleton (um objeto único), faz sentido usar `is`. De longe, o caso mais comum é verificar se uma variável está associada a `None`. A maneira recomendada de fazer isso é:

```
x is None
```

A maneira apropriada de escrever sua negação é:

```
x is not None
```

O operador `is` é mais rápido que `==` porque ele não pode ser sobrecarregado, portanto Python não precisa encontrar e chamar métodos especiais para avaliá-lo, e o processamento é tão simples quanto comparar dois IDs inteiros. Em comparação, `a == b` é um açúcar sintático para `a.__eq__(b)`. O método `__eq__` herdado de `object` compara IDs de objetos, portanto gera o mesmo resultado que `is`. Porém a maioria dos tipos embutidos sobrescreve `__eq__` com implementações mais úteis que, na prática, levam em consideração os valores dos atributos dos objetos. A igualdade pode envolver muito processamento – por exemplo, ao comparar coleções grandes ou estruturas aninhadas em vários níveis.

Para encerrar essa discussão de identidade *versus* igualdade, veremos que o tão famoso tipo `tuple` imutável não é tão rígido quanto poderíamos esperar.

A relativa imutabilidade das tuplas

As tuplas, como a maioria das coleções em Python – listas, dicionários, conjuntos etc. – armazenam referências a objetos.¹ Se os itens referenciados forem mutáveis, eles poderão mudar mesmo que a tupla em si não mude. Em outras palavras, a imutabilidade das tuplas, na verdade, refere-se ao conteúdo físico da estrutura de dados de `tuple` (isto é, às referências que ela armazena), mas não se estende aos objetos referenciados.

O exemplo 8.5 mostra a situação em que o valor de uma tupla muda como resultado de alterações em um objeto mutável referenciado por ela. O que nunca muda em uma tupla é a identidade dos itens que ela contém.

Exemplo 8.5 – `t1` e `t2` inicialmente são comparados como iguais, mas alterar um item mutável da tupla `t1` a torna diferente

```
>>> t1 = (1, 2, [30, 40]) ❶
>>> t2 = (1, 2, [30, 40]) ❷
>>> t1 == t2 ❸
True
>>> id(t1[-1]) ❹
4302515784
>>> t1[-1].append(99) ❺
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1]) ❻
4302515784
>>> t1 == t2 ❼
False
```

¹ Por outro lado, sequências de tipo único como `str`, `bytes` e `array.array` são simples: elas não contêm referências, mas armazenam fisicamente seus dados – caracteres, bytes e números – em memória contígua.

- ❶ `t1` é imutável, mas `t1[-1]` é mutável.
- ❷ Cria uma tupla `t2` cujos itens são iguais aos de `t1`.
- ❸ Embora sejam objetos distintos, `t1` e `t2` são comparados como iguais, conforme esperado.
- ❹ Inspecciona a identidade da lista em `t1[-1]`.
- ❺ Modifica a lista `t1[-1]` in-place.
- ❻ A identidade de `t1[-1]` não mudou; apenas seu valor foi alterado.
- ❼ `t1` e `t2` agora são diferentes.

Essa relativa imutabilidade das tuplas explica “O enigma da atribuição +=” na página 67. É também o motivo pelo qual algumas tuplas não são hashable, como vimos na seção “O que é Hashable?” na página 94.

A distinção entre igualdade e identidade terá outras implicações se você precisar copiar um objeto. Uma cópia é um objeto igual com um ID diferente. Porém, se um objeto contiver outros objetos, a cópia deverá também duplicar os objetos internos ou não haverá problemas em compartilhá-los? Não há uma única resposta. Continue lendo para ver uma discussão.

Cópias são rasas por padrão

A maneira mais fácil de copiar uma lista (ou a maioria das coleções embutidas mutáveis) é usar o construtor embutido do próprio tipo. Por exemplo:

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1) ❶
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1 ❷
True
>>> l2 is l1 ❸
False
```

- ❶ `list(l1)` cria uma cópia de `l1`.
- ❷ As cópias são iguais.
- ❸ Mas referem-se a dois objetos diferentes.

Para listas e outras sequências mutáveis, o atalho `l2 = l1[:]` também cria uma cópia.

No entanto usar o construtor ou `[]` produz uma *cópia rasa* (shallow copy, ou seja, a coleção mais externa é duplicada, mas a cópia é preenchida com referências aos mesmos itens armazenados na coleção original). Isso economiza memória e não causará problemas se todos os itens forem imutáveis. Contudo, se houver itens mutáveis, essa operação poderá resultar em surpresas desagradáveis.

No exemplo 8.6, criamos uma cópia rasa de uma lista que contém outra lista e uma tupla e, em seguida, fizemos alterações para ver como os objetos referenciados foram afetados.



Se você tiver um computador conectado a Internet à mão, recomendo observar a animação interativa do exemplo 8.6 em Online Python Tutor (<http://www.pythontutor.com/>). Quando escrevi este texto, um link direto com um exemplo pronto em *pythontutor.com* não estava funcionando de modo confiável, mas a ferramenta é incrível, portanto vale a pena investir tempo para copiar e colar o código.

Exemplo 8.6 – Fazendo uma cópia rasa de uma lista que contém outra lista; copie e cole este código para ver sua animação em Online Python Tutor

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)      # ❶
l1.append(100)     # ❷
l1[1].remove(55)   # ❸
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22]  # ❹
l2[2] += (10, 11)  # ❺
print('l1:', l1)
print('l2:', l2)
```

- ❶ `l2` é uma cópia rasa de `l1`. Esse estado está representado na figura 8.3.
- ❷ Concatenar `100` em `l1` não tem nenhum efeito em `l2`.
- ❸ Nesse ponto, removemos `55` da lista interna `l1[1]`. Isso afeta `l2` porque `l2[1]` está associado à mesma lista em `l1[1]`.
- ❹ Para um objeto mutável, como a lista referenciada por `l2[1]`, o operador `+=` altera a lista in-place. Essa mudança é visível em `l1[1]`, que é um apelido para `l2[1]`.
- ❺ `+=` em uma tupla cria uma nova tupla e faz a reassociação da variável `l2[2]`. É o mesmo que executar `l2[2] = l2[2] + (10, 11)`. Agora as tuplas na última posição de `l1` e de `l2` não são mais o mesmo objeto. Veja a figura 8.4.

A saída do exemplo 8.6 está no exemplo 8.7, e o estado final dos objetos está representado na figura 8.4.

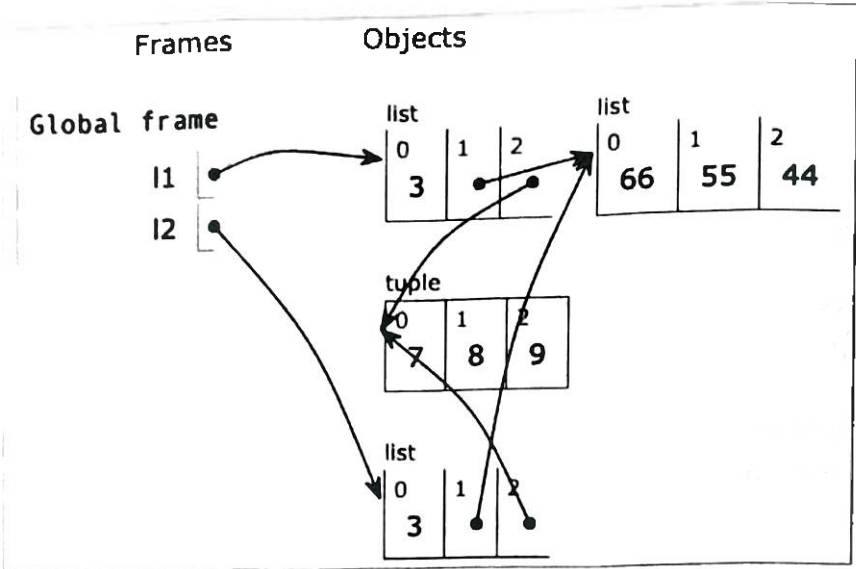


Figura 8.3 – Estado do programa imediatamente após a atribuição: `l2 = list(l1)` no exemplo 8.6. `l1` e `l2` referenciam listas distintas, mas as listas compartilham referências ao mesmo objeto `list` interno `[66, 55, 44]` e à mesma tupla `(7, 8, 9)`. (Diagrama gerado por Online Python Tutor.)

Exemplo 8.7 – Saída do exemplo 8.6

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

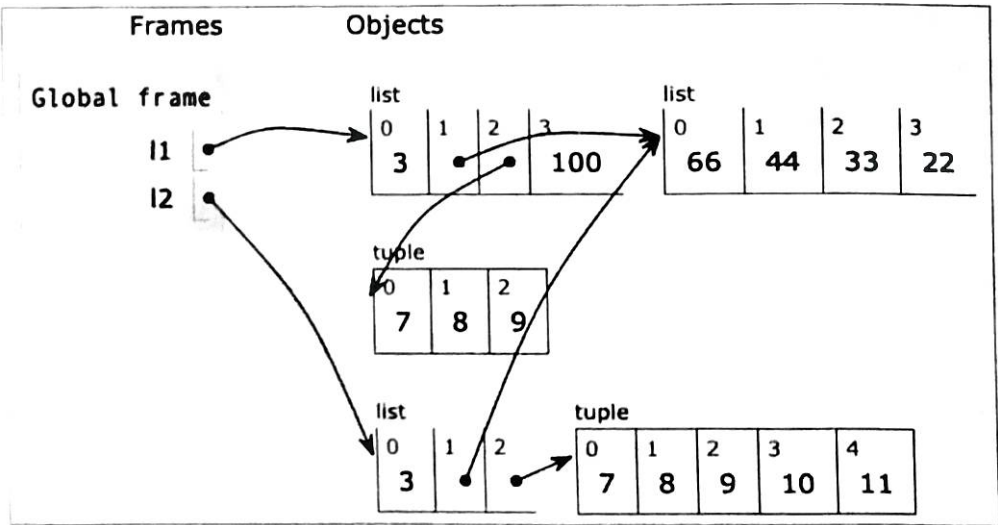


Figura 8.4 – Estado final de `l1` e `l2`: elas continuam compartilhando referências ao mesmo objeto `list`, que agora contém `[66, 44, 33, 22]`, mas a operação `l2[2] += (10, 11)` criou uma nova tupla cujo conteúdo é `(7, 8, 9, 10, 11)`, sem relação com a tupla `(7, 8, 9)` referenciada por `l1[2]`. (Diagrama gerado por Online Python Tutor.)

A essa altura, deve estar claro que cópias rasas são fáceis de fazer, mas elas podem ou não ser o que você quer. Como fazer cópias profundas será o nosso próximo assunto.

Cópias profundas e rasas de objetos quaisquer

Trabalhar com cópias rasas nem sempre é um problema, mas, às vezes, você precisará fazer cópias profundas (isto é, duplicatas que não compartilhem referências aos objetos incluídos). O módulo `copy` oferece as funções `deepcopy` e `copy`, que devolvem cópias profundas e rasas de objetos quaisquer.

Para ilustrar o uso de `copy()` e de `deepcopy()`, o exemplo 8.8 define uma classe simples, `Bus`, que representa um ônibus escolar que é carregado de passageiros e, em seguida, pega ou deixa passageiros em seu caminho.

Exemplo 8.8 – Bus pega e deixa passageiros

```
class Bus:
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)
    def pick(self, name):
        self.passengers.append(name)
    def drop(self, name):
        self.passengers.remove(name)
```

No exemplo 8.9 interativo, criaremos um objeto `bus` (`bus1`) e dois clones – uma cópia rasa (`bus2`) e uma cópia profunda (`bus3`) – para observar o que acontece quando `bus1` deixa um estudante.

Exemplo 8.9 – Efeitos do uso de `copy` versus `deepcopy`

```
>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752) ❶
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David'] ❷
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
(4302658568, 4302658568, 4302657800) ❸
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David'] ❹
```

- ❶ Usando `copy` e `deepcopy`, criamos três instâncias distintas de `Bus`.
- ❷ Depois que `bus1` deixa 'Bill', ele também fica ausente em `bus2`.
- ❸ A inspeção do atributo `passengers` mostra que `bus1` e `bus2` compartilham o mesmo objeto lista, pois `bus2` é uma cópia rasa de `bus1`.
- ❹ `bus3` é uma cópia profunda de `bus1`, portanto seu atributo `passengers` referencia outra lista.

Observe que fazer cópias profundas não é simples no caso geral. Os objetos podem ter referências cíclicas que fariam um algoritmo ingênuo entrar em um laço infinito. A função `deepcopy` lembra os objetos já copiados para tratar referências cíclicas de modo elegante. Isso é mostrado no exemplo 8.10.

Exemplo 8.10 – Referências cíclicas: `b` referencia `a` e, em seguida, é concatenado a `a`; `deepcopy` ainda é capaz de copiar `a`

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```

Além disso, uma cópia profunda pode ser profunda demais em alguns casos. Por exemplo, objetos podem fazer referência a recursos externos ou singletons que não deveriam ser copiados. Você pode controlar o comportamento tanto de `copy` quanto de `deepcopy` implementando os métodos especiais `__copy__()` e `__deepcopy__()`, conforme descritos na documentação do módulo `copy` (<http://docs.python.org/3/library/copy.html>).

O compartilhamento de objetos por meio de apelidos também explica como a passagem de parâmetros funciona em Python e o problema de usar tipos mutáveis como defaults de parâmetros. Esses problemas serão discutidos a seguir.

Parâmetros de função como referências

O único modo de passar parâmetros em Python é a *chamada por compartilhamento* (call by sharing). É o mesmo modo usado na maioria das linguagens orientadas a objetos, incluindo Ruby, SmallTalk e Java (em Java isso se aplica aos tipos de referência; tipos

primitivos usam chamada por valor naquela linguagem). Chamada por compartilhamento quer dizer que cada parâmetro formal da função obtém uma cópia de cada referência nos argumentos. Em outras palavras, os parâmetros na função tornam-se apelidos dos argumentos.

O resultado desse esquema é que uma função pode alterar qualquer objeto mutável passado como parâmetro, mas não poderá mudar a identidade desses objetos (isto é, não poderá substituir totalmente um objeto por outro). O exemplo 8.11 mostra uma função simples que usa += em um de seus parâmetros. Quando passamos números, listas e tuplas para a função, os argumentos passados são afetados de modos diferentes.

Exemplo 8.11 – Uma função pode alterar qualquer objeto mutável que ela receber

```
>>> def f(a, b):  
...     a += b  
...     return a  
...  
>>> x = 1  
>>> y = 2  
>>> f(x, y)  
3  
>>> x, y ❶  
(1, 2)  
>>> a = [1, 2]  
>>> b = [3, 4]  
>>> f(a, b)  
[1, 2, 3, 4]  
>>> a, b ❷  
([1, 2, 3, 4], [3, 4])  
>>> t = (10, 20)  
>>> u = (30, 40)  
>>> f(t, u) ❸  
(10, 20, 30, 40)  
>>> t, u  
((10, 20), (30, 40))
```

❶ O número x não é alterado.

❷ A lista a é alterada.

❸ A tupla t não é alterada.

Outro problema relacionado a parâmetros de função está no uso de valores mutáveis para defaults, conforme discutiremos a seguir.

Tipos mutáveis como default de parâmetros: péssima ideia

Parâmetros opcionais com valores default são um ótimo recurso de definições de função em Python, permitindo que nossas APIs evoluam, ao mesmo tempo que permanecem compatíveis com versões anteriores. Porém você deve evitar objetos mutáveis como valores default para parâmetros.

Para ilustrar essa questão, no exemplo 8.12, usamos a classe `Bus` do exemplo 8.8 e alteramos seu método `__init__` para criar `HauntedBus` – um ônibus mal-assombrado. Nesse caso, tentamos ser espertos e, em vez de ter um valor default `passengers=None`, temos `passengers=[]`, evitando assim o `if` do `__init__` anterior. Essa “esperteza” causa problemas.

Exemplo 8.12 – Uma classe simples para ilustrar o perigo de um default mutável

```
class HauntedBus:
    """Um modelo de ônibus assombrado por passageiros fantasmas"""
    def __init__(self, passengers=[]): ❶
        self.passengers = passengers ❷
    def pick(self, name):
        self.passengers.append(name) ❸
    def drop(self, name):
        self.passengers.remove(name)
```

- ❶ Quando o argumento `passengers` não é passado, esse parâmetro é associado ao objeto lista default, que, inicialmente, é uma lista vazia.
- ❷ Essa atribuição faz de `self.passengers` um apelido para `passengers`, que, por sua vez, é um apelido para a lista default quando nenhum argumento `passengers` é especificado.
- ❸ Quando os métodos `.remove()` e `.append()` são usados com `self.passengers`, na verdade, estamos mudando a lista default, que é um atributo do objeto-função.

O exemplo 8.13 mostra o comportamento assustador de `HauntedBus`.

Exemplo 8.13 – Ônibus assombrados por passageiros fantasmas

```
>>> bus1 = HauntedBus(['Alice', 'Bill'])
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers ❶
['Bill', 'Charlie']
>>> bus2 = HauntedBus() ❷
>>> bus2.pick('Carrie')
```

```

>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus() ❸
>>> bus3.passengers ❹
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers ❺
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers ❻
True
>>> bus1.passengers ❼
['Bill', 'Charlie']

```

- ❶ Até agora, tudo bem: nenhuma surpresa com `bus1`.
- ❷ `bus2` começa vazio, portanto a lista vazia default é atribuída a `self.passengers`.
- ❸ `bus3` também começa vazio; novamente, a lista default é atribuída.
- ❹ O default não está mais vazio!
- ❺ Agora Dave, que subiu em `bus3`, aparece em `bus2`.
- ❻ O problema: `bus2.passengers` e `bus3.passengers` referenciam a mesma lista.
- ❼ Mas `bus1.passengers` é uma lista distinta.

O problema é que as instâncias de `HauntedBus` que não recebem uma lista inicial de passageiros acabam compartilhando a mesma lista de passageiros entre si.

Esse tipo de bug pode ser sutil. Como mostra o exemplo 8.13, quando um `HauntedBus` é instanciado com passageiros, ele funciona conforme esperado. Coisas estranhas acontecem somente quando um `HauntedBus` começa vazio, pois `self.passengers` passa a ser um apelido para o valor default do parâmetro `passengers`. O problema é que cada valor default é avaliado quando a função é definida – isto é, normalmente, quando o módulo é carregado – e os valores default tornam-se atributos do objeto-função. Sendo assim, se um valor default for um objeto mutável e você alterá-lo, a mudança afetará todas as chamadas daquela função no futuro.

Após executar as linhas do exemplo 8.13, você poderá inspecionar o objeto `HauntedBus.__init__` e ver os estudantes fantasmas assombrando o atributo `__defaults__` desse objeto:

```

>>> dir(HauntedBus.__init__) # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'],)

```


Por fim, podemos verificar se `bus2.passengers` é um apelido associado ao primeiro elemento do atributo `HauntedBus.__init__.__defaults__`:

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

O problema com `defaults` mutáveis explica por que `None` geralmente é usado como o valor default de parâmetros que possam receber valores mutáveis. No exemplo 8.8, `__init__` verifica se o argumento `passengers` é `None` e atribui uma nova lista vazia a `self.passengers`. Como explicado na seção a seguir, se `passengers` for diferente de `None`, a implementação correta atribuirá uma cópia dele a `self.passengers`. Vamos agora dar uma olhada com mais detalhes.

Programação defensiva com parâmetros mutáveis

Ao escrever uma função que receba um parâmetro mutável, você deve considerar cuidadosamente se quem a chama espera que o argumento passado seja alterado.

Por exemplo, se sua função recebe um `dict` e precisa modificá-lo enquanto o processa, esse efeito colateral deverá ser visível fora da função? Na verdade, isso depende do contexto. É realmente uma questão de alinhar a expectativa de quem escreve a função com a de quem a chama.

O último exemplo do ônibus neste capítulo mostra como um `TwilightBus` vai contra as expectativas ao compartilhar sua lista de passageiros com seus clientes. Antes de estudar a implementação, observe o exemplo 8.14 para ver como a classe `TwilightBus` funciona do ponto de vista de um cliente da classe.

Exemplo 8.14 – Passageiros desaparecem quando deixados por um `TwilightBus`

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat'] ❶
>>> bus = TwilightBus(basketball_team) ❷
>>> bus.drop('Tina') ❸
>>> bus.drop('Pat')
>>> basketball_team ❹
['Sue', 'Maya', 'Diana']
```

- ❶ `basketball_team` armazena cinco nomes de estudantes.
- ❷ Um `TwilightBus` é carregado com a equipe.
- ❸ `bus` deixa uma estudante e, em seguida, deixa outra.
- ❹ As passageiras deixadas sumiram do time de basquete!

TwilightBus viola o “Princípio da mínima surpresa” (Principle of least astonishment), que é uma boa prática no design de interfaces. Quando o ônibus deixa uma estudante, a remoção de seu nome da equipe de basquete pode ser uma surpresa desagradável. O exemplo 8.15 mostra a implementação de TwilightBus e uma explicação para o problema.

Exemplo 8.15 – Uma classe simples que mostra os perigos de alterar argumentos recebidos

```
class TwilightBus:
    """Um modelo de ônibus que faz os passageiros desaparecerem"""
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = [] ❶
        else:
            self.passengers = passengers ❷
    def pick(self, name):
        self.passengers.append(name)
    def drop(self, name):
        self.passengers.remove(name) ❸
```

- ❶ Nesse caso, tomamos cuidado para criar uma nova lista vazia quando `passengers` é `None`.
- ❷ No entanto essa atribuição faz de `self.passengers` um apelido para `passengers`, que, por sua vez, é um apelido para o verdadeiro argumento passado para `__init__` (ou seja, `basketball_team` no exemplo 8.14).
- ❸ Quando os métodos `.remove()` e `.append()` são usados com `self.passengers`, na verdade, estamos alterando a lista original recebida como argumento do construtor.

O problema, nesse caso, é que o ônibus está usando um apelido para a lista passada para o construtor. Em vez disso, ele deveria manter sua própria lista de passageiros. A correção é simples: em `__init__`, quando o parâmetro `passengers` é fornecido, `self.passengers` deve ser inicializado com uma cópia do parâmetro, como fizemos corretamente no exemplo 8.8 (seção “Cópias profundas e rasas de objetos quaisquer” na página 267):

```
def __init__(self, passengers=None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers) ❶
```

- ❶ Faz uma cópia da lista `passengers` ou converte o argumento em uma `list`, se ele não for uma lista.

Agora, o nosso tratamento interno da lista de passageiros não afetará o argumento usado para inicializar o ônibus. Como bônus, essa solução é mais flexível: agora o argumento passado para o parâmetro `passengers` pode ser um `tuple` ou qualquer outro iterável, como um `set` ou até mesmo resultados de uma consulta ao banco de dados, pois o construtor `list()` aceita qualquer iterável. Ao criar nossa própria lista para administrarmos, garantimos que ela implementa as operações `.remove()` e `.append()` necessárias aos métodos `.pick()` e `.drop()`



A menos que o propósito de um método seja explicitamente alterar um objeto recebido como argumento, você deve pensar duas vezes antes de usar apelidos no objeto recebido como argumento simplesmente atribuindo-o a uma variável de instância em sua classe. Na dúvida, faça uma cópia. Seus clientes geralmente ficarão mais satisfeitos.

del e coleta de lixo

Os objetos jamais são explicitamente destruídos; no entanto, quando se tornarem inacessíveis, eles poderão ser eliminados pela coleta de lixo.

— capítulo “Data Model” (Modelo de dados) de *The Python Language Reference*

O comando `del` apaga nomes, e não objetos. Um objeto pode ser destruído pela coleta de lixo como resultado de um comando `del`, mas somente se a variável apagada armazenar a última referência ao objeto ou se o objeto se tornar inacessível.² A reassociação de uma variável também pode reduzir a zero o número de referências a um objeto, causando a sua destruição.



Há um método especial `__del__`, mas ele não elimina a instância e não deve ser chamado pelo seu código. `__del__` é chamado pelo interpretador Python quando a instância está prestes a ser destruída para que ela tenha uma chance de liberar recursos externos. Raramente você precisará implementar `__del__` em seu código, apesar de alguns iniciantes em Python gastarem tempo implementando-o sem que haja um bom motivo. É complicado usar corretamente `__del__`. Consulte a documentação do método especial `__del__` (<http://bit.ly/1GsWPac>) no capítulo “Data Model” de *The Python Language Reference* (Guia de referência à linguagem Python).

Em CPython, o algoritmo principal para coleta de lixo é a contagem de referências. Essencialmente, cada objeto mantém um contador para o número de referências a ele. Assim que esse *refcount* atinge zero, o objeto é imediatamente destruído: CPython chama o método `__del__` do objeto (se estiver definido) e libera a memória alocada a esse objeto. Em CPython 2.0, um algoritmo geracional de coleta de lixo geracional

² Se dois objetos referenciarem um ao outro, como no exemplo 8.10, eles poderão ser destruídos caso o coletor de lixo determine que eles sejam inacessíveis porque suas únicas referências são as referências mútuas.

(generational garbage collector) foi implementado para detectar grupos de objetos envolvidos em referências cíclicas – que poderão ser inacessíveis, mesmo com referências explícitas a eles, quando todas as referências mútuas estiverem contidas no grupo. Outras implementações de Python têm coletores de lixo mais sofisticados, que não dependem de contagem de referências, o que significa que o método `__del__` poderá não ser chamado imediatamente quando não houver mais referências ao objeto. Acesse “PyPy, Garbage Collection, and a Deadlock” (PyPy, coleta de lixo e um deadlock, <http://bit.ly/1GsWTa7>) de A. Jesse Jiryu Davis para ver uma discussão sobre o uso próprio e impróprio de `__del__`.

Para mostrar o fim da vida de um objeto, o exemplo 8.16 usa `weakref.finalize` para registrar uma função de callback a ser chamada quando um objeto é destruído.

Exemplo 8.16 – Observando o fim de um objeto quando não há mais referências a ele

```
>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1 ❶
>>> def bye(): ❷
...     print('Gone with the wind...')
...
>>> ender = weakref.finalize(s1, bye) ❸
>>> ender.alive ❹
True
>>> del s1
>>> ender.alive ❺
True
>>> s2 = 'spam' ❻
Gone with the wind...
>>> ender.alive
False
```

- ❶ `s1` e `s2` são apelidos que se referem ao mesmo conjunto `{1, 2, 3}`.
- ❷ Essa função não deve ser um método associado do objeto prestes a ser destruído ou que armazene uma referência a ele.
- ❸ Registra a callback `bye` no objeto referenciado por `s1`.
- ❹ O atributo `.alive` é `True` antes de o objeto `finalize` ser chamado.
- ❺ Conforme discutido, `del` não apaga um objeto, mas apenas uma referência a ele.
- ❻ A reassociação da última referência, ou seja, de `s2`, deixa `{1, 2, 3}` inacessível. Ele será destruído, a callback `bye` será chamada e `ender.alive` se tornará `False`.

O ponto principal no exemplo 8.16 é deixar explícito que `del` não apaga objetos, porém eles podem ser apagados como consequência de se tornarem inacessíveis após o uso de `del`.

Você pode estar se perguntando por que o objeto `{1, 2, 3}` foi destruído no exemplo 8.16. Afinal de contas, a referência `s1` foi passada para a função `finalize`, que deve ter contado com ela para monitorar o objeto e chamar a callback. Isso funciona porque `finalize` armazena uma *referência fraca* (weak reference) para `{1, 2, 3}`, conforme explicado na próxima seção.

Referências fracas

A presença de referências é o que mantém um objeto vivo na memória. Quando o contador de referências de um objeto for zero, o coletor de lixo destruirá o objeto. Às vezes, porém, é conveniente ter uma referência a um objeto que não o mantenha por mais tempo que o necessário. Um caso de uso comum é um cache.

Referências fracas a um objeto não incrementam o seu contador de referências. O objeto-alvo de uma referência é chamado de *referente*. Sendo assim, dizemos que uma referência fraca não impede que o referente seja destruído pelo coletor de lixo.

Referências fracas são úteis em aplicações de caching porque você não vai querer que objetos em cache permaneçam vivos somente porque são referenciados pelo cache.

O exemplo 8.17 mostra como uma instância de `weakref.ref` pode ser chamada para acessar seu referente. Se o objeto estiver vivo, chamar a referência fraca devolve esse objeto; caso contrário, `None` será devolvido.



O exemplo 8.17 é uma sessão de console, e o console de Python associa automaticamente a variável `_` ao resultado de expressões que não sejam `None`. Isso interferiu na demonstração que eu pretendia fazer, mas também enfatizou uma questão prática: quando tentamos microgerenciar a memória, com frequência, somos surpreendidos por atribuições ocultas e implícitas que criam novas referências aos nossos objetos. A variável de console `_` é um exemplo. Objetos `traceback` são outra fonte comum de referências inesperadas.

Exemplo 8.17 – Uma referência fraca é um invocável que devolve o objeto referenciado ou `None` se o referente não existir

```
>>> import weakref
>>> a_set = {0, 1}
>>> wref = weakref.ref(a_set) ❶
>>> wref
```

```

<weakref at 0x100637598; to 'set' at 0x100636748>
>>> wref() ❷
{0, 1}
>>> a_set = {2, 3, 4} ❸
>>> wref() ❹
{0, 1}
>>> wref() is None ❺
False
>>> wref() is None ❻
True

```

- ❶ O objeto de referência fraca `wref` é criado e inspecionado na próxima linha.
- ❷ Chamar `wref()` devolve o objeto referenciado, ou seja, `{0, 1}`. Como essa é uma sessão de console, o resultado `{0, 1}` está associado à variável `_`.
- ❸ `a_set` não se refere mais ao conjunto `{0, 1}`, portanto seu contador de referências é decrementado. Porém a variável `_` continua referenciando-o.
- ❹ Chamar `wref()` continua devolvendo `{0, 1}`.
- ❺ Quando essa expressão é avaliada, `{0, 1}` está vivo, portanto `wref()` não é `None`. Mas então `_` é associado ao valor resultante, `False`. Agora não há mais referências fortes a `{0, 1}`.
- ❻ Como o objeto `{0, 1}` não existe mais, essa última chamada a `wref()` devolve `None`.

A documentação do módulo `weakref` (<https://docs.python.org/3/library/weakref.html>) destaca que a classe `weakref.ref`, na verdade, é uma interface de baixo nível, criada para usuários avançados, e que a maioria dos programas estará mais bem servida se usar coleções de `weakref` e finalize. Em outras palavras, considere o uso de `WeakKeyDictionary`, `WeakValueDictionary`, `WeakSet` e `finalize` (que utilizam referências fracas internamente) em vez de criar e tratar suas próprias instâncias de `weakref.ref` manualmente. Fizemos isso no exemplo 8.17 somente porque mostrar um único `weakref.ref` em ação elimina parte do mistério em torno deles. Na prática, porém, na maior parte do tempo, os programas Python usam as coleções de `weakref`.

A próxima subseção discutirá rapidamente as coleções de `weakref`.

Esquete com `WeakValueDictionary`

A classe `WeakValueDictionary` implementa um mapeamento mutável em que os valores são referências fracas a objetos. Quando um objeto referenciado é destruído pelo coletor de lixo em outro ponto do programa, a chave correspondente é automaticamente removida de `WeakValueDictionary`. Isso é comumente usado em caching.

Nossa demonstração de um `WeakValueDictionary` é inspirada no clássico esquete *Cheese Shop* (A loja de queijos) de Monty Python, em que um cliente pede mais de 40 tipos de queijo, incluindo *cheddar* e *mozzarella*, mas não há nenhum em estoque.³

O exemplo 8.18 implementa uma classe simples que representa cada tipo de queijo.

Exemplo 8.18 – Cheese tem um atributo `kind` e uma representação-padrão

```
class Cheese:
    def __init__(self, kind):
        self.kind = kind

    def __repr__(self):
        return 'Cheese(%r)' % self.kind
```

No exemplo 8.19, cada queijo é carregado de um `catalog` para um `stock` implementado como um `WeakValueDictionary`. No entanto todos desaparecem de `stock`, exceto um, assim que `catalog` é apagado. Você é capaz de explicar por que o queijo parmesão dura mais que os outros?⁴ A dica após o código tem a resposta.

Exemplo 8.19 – Cliente: “Final de contas, você tem algum tipo de queijo aqui?”

```
>>> import weakref
>>> stock = weakref.WeakValueDictionary() ❶
>>> catalog = [Cheese('Red Leicester'), Cheese('Tilsit'),
...             Cheese('Brie'), Cheese('Parmesan')]
>>> for cheese in catalog:
...     stock[cheese.kind] = cheese ❷
>>> sorted(stock.keys())
['Brie', 'Parmesan', 'Red Leicester', 'Tilsit'] ❸
>>> del catalog
>>> sorted(stock.keys())
['Parmesan'] ❹
>>> del cheese
>>> sorted(stock.keys())
[]
```

3 `cheeseshop.python.org` é também um apelido para PyPI – o repositório de software Python Package Index (Índice de pacotes Python) –, que iniciou sua vida quase vazio. Quando escrevi este capítulo, havia 41.426 pacotes no Python Cheese Shop. Nada mal, mas ainda distante dos mais de 131 mil módulos disponíveis em CPAN – Comprehensive Perl Archive Network –, invejado por todas as comunidades de linguagens dinâmicas.

4 O queijo parmesão é envelhecido durante pelo menos um ano na fábrica, portanto é mais durável que um queijo fresco, mas essa não é a resposta que estamos procurando.

- ❶ stock é um `WeakValueDictionary`.
- ❷ stock mapeia o nome do queijo a uma referência fraca para a instância do queijo em `catalog`.
- ❸ stock está completo.
- ❹ Depois que `catalog` é apagado, a maioria dos queijos não existe mais em `stock`, como esperado em `WeakValueDictionary`. Por que isso não ocorre com todos os queijos, nesse caso?



Uma variável temporária pode fazer um objeto durar mais que o esperado ao armazenar uma referência a ele. Normalmente, isso não é um problema com variáveis locais: elas são destruídas no retorno da função. Contudo, no exemplo 8.19, a variável `cheese` do laço `for` é uma variável global e jamais desaparecerá, a menos que seja explicitamente apagada.

Uma contrapartida de `WeakValueDictionary` é `WeakKeyDictionary`, em que as chaves são referências fracas. A documentação de `weakref.WeakKeyDictionary` (<http://bit.ly/1GsXB6Z>) dá dicas sobre possíveis usos:

[Um `WeakKeyDictionary`] pode ser usado para associar dados adicionais a um objeto pertencente a outras partes de uma aplicação sem acrescentar atributos a esses objetos. Isso pode ser útil, em especial, com objetos que sobrescrevem métodos de acesso a atributos.

O módulo `weakref` também oferece um `WeakSet`, descrito de modo simples na documentação como “classe de conjunto que mantém referências fracas aos seus elementos. Um elemento será descartado quando não houver mais nenhuma referência forte a ele.” Se você precisar criar uma classe que deva ter conhecimento de todas as suas instâncias, uma boa solução será criar um atributo de classe com um `WeakSet` para armazenar as referências às instâncias. Do contrário, se um `set` normal fosse usado, as instâncias jamais seriam destruídas com o coletor de lixo porque a própria classe teria referências fortes a elas, e as classes vivem tanto quanto o processo Python, a menos que você as apague deliberadamente.

Essas coleções, e as referências fracas em geral, são limitadas quanto aos tipos de objeto que elas podem manipular. A próxima seção explica isso.

Limitações das referências fracas

Nem todo objeto Python pode ser o alvo – ou o referente – de uma referência fraca. Instâncias de `list` e `dict` básicos não podem ser referentes, mas uma subclasse simples de qualquer um deles pode resolver esse problema facilmente:

```
class MyList(list):
    """subclasse de list cujas instâncias podem ter referências fracas"""

a_list = MyList(range(10))
# a_list pode ser alvo de uma referência fraca
wref_to_a_list = weakref.ref(a_list)
```

Uma instância de set pode ser um referente, e é por isso que um set foi usado no exemplo 8.17. Tipos definidos pelo usuário também não apresentam problemas, o que explica por que a classe singela Cheese foi necessária no exemplo 8.19. Entretanto instâncias de int e de tuple não podem ser alvos de referências fracas, mesmo que subclasses desses tipos sejam criadas.

A maioria dessas limitações são detalhes de implementação de CPython e podem não se aplicar a outros interpretadores Python. Elas resultam de otimizações internas; algumas das quais serão discutidas na próxima seção, que é bastante opcional.

Truques de Python com imutáveis



Você pode tranquilamente pular esta seção. Ela discute alguns detalhes de implementação de Python que não são realmente importantes para *usuários* de Python. São atalhos e otimizações feitos pelos core developers de CPython que não deverão incomodar você no uso da linguagem e poderão não se aplicar a outras implementações de Python, nem sequer em futuras versões de CPython. Apesar disso, ao fazer testes com apelidos e cópias, talvez você acabe se deparando com esses truques, portanto achei que vale a pena mencioná-los.

Fiquei surpreso ao saber que, para uma tupla `t`, `t[:]` não cria uma cópia, mas devolve uma referência ao mesmo objeto. Você também obterá uma referência à mesma tupla se usar `tuple(t)`.⁵ O exemplo 8.20 comprova isso.

Exemplo 8.20 – Uma tupla criada a partir de outra, na verdade, é exatamente a mesma tupla

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1 ❶
True
>>> t3 = t1[:]
>>> t3 is t1 ❷
True
```

⁵ Isso está claramente documentado. Digite `help(tuple)` no console de Python e você verá: “If the argument is a tuple, the return value is the same object.” (Se o argumento for uma tupla, o valor de retorno será o mesmo objeto.) Achei que sabia tudo sobre tuplas até escrever este livro.

❶ t1 e t2 estão associados ao mesmo objeto.

❷ Assim como t3.

O mesmo comportamento pode ser observado com instâncias de `str`, `bytes` e `frozenset`. Observe que um `frozenset` não é uma sequência, portanto `fs[:]` não funcionará se `fs` for um `frozenset`. Entretanto `fs.copy()` tem o mesmo efeito: ele trapaceia e devolve uma referência ao mesmo objeto, e não uma cópia, como mostra o exemplo 8.21.⁶

Exemplo 8.21 – Strings literais podem criar objetos compartilhados

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3) # ❶
>>> t3 is t1 # ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC' # ❸
>>> s2 is s1 # ❹
True
```

❶ Criando uma nova tupla do zero.

❷ t1 e t3 são iguais, mas não são o mesmo objeto.

❸ Criando um segundo `str` do zero.

❹ Surpresa: a e b referem-se ao mesmo `str`!

O compartilhamento de strings literais é uma técnica de otimização chamada *internalização* (interning). CPython usa a mesma técnica com números inteiros pequenos para evitar a duplicação desnecessária de objetos representando números “populares” como 0, -1 e 42. Observe que CPython não internaliza todas as strings ou todos os inteiros, e os critérios que ele usa para isso são detalhes de implementação não documentados.



Jamais conte com a internalização de `str` ou de `int`! Sempre use `==` em vez de `is` para comparar se são iguais. A internalização é um recurso para uso interno do interpretador Python.

Os truques discutidos nesta seção, incluindo o comportamento de `frozenset.copy()`, são “mentirinhas”; elas economizam memória e deixam o interpretador mais rápido. Não se preocupe; elas não causarão nenhum problema a você, pois se aplicam somente a tipos imutáveis. Provavelmente, essas minúcias só lhe servirão para vencer apostas com colegas Pythonistas.

⁶ A “mentirinha” de ter um método `copy` que não copia nada pode ser explicada pela compatibilidade de interface: isso faz `frozenset` ser mais compatível com `set`. De qualquer modo, o fato de dois objetos imutáveis idênticos serem iguais ou serem cópias não faz diferença alguma para o usuário final.

Resumo do capítulo

Todo objeto Python tem uma identidade, um tipo e um valor. Somente o valor de um objeto pode mudar ao longo do tempo.⁷

Se duas variáveis referenciam objetos imutáveis com valores iguais (`a == b` é `True`), na prática, raramente importa se elas referenciam cópias ou se são apelidos referenciando o mesmo objeto porque o valor de um objeto imutável não muda, com uma exceção. A exceção são as coleções imutáveis como tuplas e frozensets: se uma coleção imutável armazenar referências a itens mutáveis, seu valor poderá mudar quando o valor de um item mutável mudar. Na prática, esse cenário não é tão comum. O que nunca muda em uma coleção imutável são as identidades dos objetos contidos.

O fato de as variáveis armazenarem referências tem várias consequências práticas em programação Python:

- Atribuições simples não criam cópias.
- A atribuição combinada com `+=` ou `*=` cria novos objetos se a variável à esquerda estiver associada a um objeto imutável, mas poderá modificar um objeto mutável in-place.
- Atribuir um novo valor a uma variável existente não muda o objeto anteriormente associado a ela. Isso se chama reassociação (rebinding); a variável agora estará associada a um objeto diferente. Se essa variável for a última referência ao objeto anterior, esse objeto será destruído pelo coletor de lixo.
- Os parâmetros de função são passados como apelidos, o que significa que a função pode alterar qualquer objeto mutável recebido como argumento. Não há como impedir isso, exceto criando cópias locais ou usando objetos imutáveis (por exemplo, passando uma tupla em vez de uma lista).
- Usar objetos mutáveis como valores default para parâmetros de função é perigoso porque, se os parâmetros forem alterados in-place, o default será alterado, afetando todas as futuras chamadas que dependam do default.

Em CPython, os objetos são descartados assim que o número de referências a eles atinge zero. Eles também podem ser descartados se formarem grupos com referências cíclicas, mas sem referências externas. Em algumas situações, pode ser útil armazenar uma referência a um objeto que – que por si só – não manterá um objeto vivo. Um exemplo é uma classe que queira manter um controle de todas as suas instâncias no momento. Isso pode ser feito com referências fracas, um mecanismo de baixo nível subjacente às coleções `WeakValueDictionary`, `WeakKeyDictionary`, `WeakSet` e a função `finalize` do módulo `weakref`, que são mais úteis que as referências fracas individuais.

⁷ Na verdade, o tipo de um objeto pode ser alterado simplesmente associando uma classe diferente ao seu atributo `__class__`, mas isso é pura maldade, e já me arrependi de ter escrito esta nota de rodapé.

Leituras complementares

O capítulo “Data Model” (Modelo de dados, <http://bit.ly/1GsZwss>) de *The Python Language Reference* (Guia de referência à linguagem Python) começa com uma explicação clara sobre identidades e valores de objetos.

Wesley Chun, autor da série de livros *Core Python*, fez uma ótima apresentação sobre muitos dos assuntos discutidos neste capítulo durante a OSCON 2013. Você pode fazer download dos slides da página da palestra “Python 103: Memory Model & Best Practices” (Python avançado: modelo de memória & melhores práticas, <http://bit.ly/1GsZvEO>). Há também um vídeo no YouTube (<http://bit.ly/1HGCayS>) de uma apresentação mais longa feita por Wesley na EuroPython 2011, abordando não só o tema deste capítulo mas também o uso de métodos especiais.

Doug Hellmann escreveu uma longa série de posts de blog excelentes chamada *Python Module of the Week* (Módulo Python da semana, <http://pymotw.com/>), que se transformou em um livro: *The Python Standard Library by Example* (<http://bit.ly/py-libex>). Seus posts “copy – Duplicate Objects” (copy – Duplicar objetos, <http://pymotw.com/2/copy/>) e “weakref – Garbage-Collectable References to Objects” (weakref – Referências a objetos que podem ser destruídas pelo coletor de lixo, <http://pymotw.com/2/weakref/>) abordam alguns dos assuntos que acabamos de discutir.

Mais informações sobre o coletor de lixo geracional de CPython podem ser encontradas na documentação do módulo gc (<http://bit.ly/1HGCbmj>), que começa com a frase “Este módulo oferece uma interface ao coletor de lixo opcional.” O adjetivo “opcional”, nesse caso, pode causar surpresa, mas o capítulo “Data Model” (Modelo de dados, <http://bit.ly/1GsZwss>) também afirma o seguinte:

Uma implementação pode adiar a coleta de lixo ou omiti-la totalmente – como a coleta de lixo é implementada é uma questão de qualidade de implementação, desde que nenhum objeto ainda acessível seja coletado.

Fredrik Lundh – criador de bibliotecas importantes como ElementTree, Tkinter e a biblioteca de imagens PIL – tem um pequeno post chamado “How Does Python Manage Memory?” (Como Python administra a memória?, <http://bit.ly/1FSDBpM>) sobre o coletor de lixo de Python. Ele enfatiza que o coletor de lixo é um recurso de implementação que se comporta de modo diferente entre interpretadores Python distintos. Por exemplo, Jython usa o coletor de lixo de Java.

O coletor de lixo de CPython 3.4 melhorou o tratamento de objetos que têm um método `__del__`, conforme descrito na PEP 442 – *Safe object finalization* (Finalização segura de objetos, <http://bit.ly/1HGCde7>).

A Wikipedia tem um artigo sobre *internalização de string* (string interning, <http://bit.ly/1HGCduC>) que menciona o uso dessa técnica em diversas linguagens, incluindo Python.

Ponto de vista

Tratamento igual a todos os objetos

Conheci Java antes de descobrir a linguagem Python. O operador `==` em Java nunca me pareceu correto. É muito mais comum que os programadores estejam interessados em igualdade em vez de identidade, mas, para objetos (não para tipos primitivos), o operador `==` de Java compara referências, e não valores de objetos. Mesmo para algo tão básico quanto comparar strings, Java força você a usar o método `.equals`. Mesmo assim, há outra armadilha: se escrever `a.equals(b)` e `a` for `null`, você obterá uma exceção de ponteiro nulo. Os designers de Java acharam necessário sobrecarregar `+` para strings, então por que não seguir em frente e sobrecarregar `==` também?

A linguagem Python resolve isso melhor. O operador `==` compara valores de objetos, enquanto `is` compara referências. Como Python tem sobrecarga de operadores, `==` funciona de forma sensata com todos os objetos da biblioteca-padrão, incluindo `None`, que é um objeto de verdade, diferente de `null` em Java.

É claro que você pode definir `__eq__` em suas próprias classes para decidir o que `==` quer dizer para suas instâncias. Se você não sobrescrever `__eq__`, o método herdado de `object` comparará IDs de objetos, portanto a norma geral é que toda instância de uma classe definida pelo usuário é considerada diferente.

Esses foram alguns dos motivos que me fizeram mudar de Java para Python assim que acabei de ler o Tutorial de Python em uma tarde em setembro de 1998.

Mutabilidade

Este capítulo seria redundante se todos os objetos fossem imutáveis. Ao lidar com objetos que não mudam, o fato de as variáveis armazenarem os objetos propriamente ditos ou referências a objetos compartilhados não faz diferença. Se `a == b` for verdadeiro e nenhum dos dois objetos puder mudar, eles podem muito bem ser o mesmo. É por isso que a internalização de strings é segura. A identidade dos objetos torna-se importante somente quando os objetos são mutáveis.

Em programação funcional “pura”, todos os dados são imutáveis: concatenar dados em uma coleção, na verdade, cria uma nova coleção. Python, porém, não é uma linguagem funcional, muito menos uma linguagem funcional pura. As instâncias de classes definidas pelo usuário são mutáveis por padrão em Python – assim como na maioria das linguagens orientadas a objetos. Ao criar seus próprios objetos, você deverá tomar um cuidado extra para torná-los imutáveis

se isso for um requisito. Todos os atributos do objeto também devem ser imutáveis; caso contrário, você acabará com algo parecido com `tuple`: imutável no que diz respeito aos IDs dos objetos, porém o valor de um `tuple` poderá mudar se ele armazenar um objeto mutável.

Objetos mutáveis também são o principal motivo pelo qual a programação com threads é tão difícil de ser feita corretamente: threads que alteram objetos sem uma sincronização apropriada produzem dados corrompidos. A sincronização excessiva, por outro lado, causa deadlocks.

Destruição de objetos e coleta de lixo

Não há nenhum mecanismo em Python para destruir diretamente um objeto, e essa omissão, na verdade, é uma grande vantagem: se você pudesse destruir um objeto a qualquer momento, o que aconteceria com referências fortes que apontassem para ele?

A coleta de lixo em CPython é feita principalmente por contagem de referências, que é fácil de implementar, mas é suscetível a vazamento de memória (memory leaking) quando houver referências cíclicas; por isso, na versão 2.0 (outubro de 2000), um coletor de lixo geracional foi implementado, capaz de destruir objetos inacessíveis que antes permaneciam vivos por causa de referências cíclicas.

Apesar disso, a contagem de referências continua presente como base e provoca a destruição imediata de objetos sem nenhuma referência. Isso quer dizer que, em CPython – pelo menos por enquanto – é seguro escrever:

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

Esse código é seguro porque o contador de referências do objeto arquivo será zero após o método `write` retornar, e Python fechará imediatamente o arquivo antes de destruir o objeto que o representa na memória. Contudo a mesma linha não é segura em Jython ou IronPython, que usam o coletor de lixo dos runtimes de seus hosts (Java VM e .NET CLR), que são mais sofisticados, porém não dependem de contagem de referências e podem demorar mais para destruir o objeto e fechar o arquivo. Em todos os casos, incluindo CPython, a melhor prática é fechar explicitamente o arquivo, e a maneira mais confiável de fazer isso é usando o comando `with`, que garante que o arquivo será fechado mesmo que exceções sejam levantadas enquanto ele estiver aberto. Com `with`, o trecho de código anterior fica assim:

```
with open('test.txt', 'wt', encoding='utf-8') as fp:  
    fp.write('1, 2, 3')
```


Se você estiver interessado no assunto coletores de lixo, leia o artigo “Python Garbage Collector Implementations: CPython, PyPy and GaS” (Implementação de coletor de lixo em Python: CPython, PyPy e GaS, <http://bit.ly/1Gt0Hrf>) de Thomas Perl, onde aprendi esse detalhe de segurança de `open().write()` em CPython.

Passagem de parâmetros: chamada por compartilhamento

Uma maneira popular de explicar como a passagem de parâmetros funciona em Python é a frase: “Parâmetros são passados por valor, mas os valores são referências.” Não está errado, mas causa confusão porque os modos mais comuns de passagem de parâmetro em linguagens mais antigas são *chamada por valor* (a função obtém uma cópia do argumento) e *chamada por referência* (a função obtém um ponteiro para o argumento). Em Python, a função obtém uma cópia dos argumentos, mas os argumentos são sempre referências. Sendo assim, o valor de cada objeto referenciado pode ser alterado se esses objetos forem mutáveis, mas suas identidades não poderão ser alteradas. Além do mais, como a função obtém uma cópia da referência em um argumento, sua reassociação não terá efeito algum fora da função. Adotei o termo *chamada por compartilhamento* (call by sharing) após estudar o assunto no livro *Programming Language Pragmatics*, 3ª edição, de Michael L. Scott (Morgan Kaufmann), particularmente a seção “8.3.1: Parameter Modes”.

A citação completa de Alice e a canção do Cavaleiro

Adoro essa passagem, mas ela era muito longa para abrir o capítulo. Eis o diálogo completo sobre a canção do Cavaleiro, o nome dela e como a canção e seu nome são chamados:

‘Você está triste’, disse o Cavaleiro ansioso, ‘deixe-me cantar uma canção para consolá-la’.

‘É muito comprida?’, perguntou Alice, pois já havia escutado um bocado de poesia naquele dia.

‘É comprida’, disse o Cavaleiro, ‘mas é muito, MUITO bonita. Todos que me ouvem cantá-la... ficam com LÁGRIMAS nos olhos, ou...’

‘Ou o quê?’, disse Alice, pois o Cavaleiro fizera uma pausa repentina.

‘Ou não ficam, sabe como é. O nome da canção é chamado “OLHOS DE HADOQUE”’

‘Ah, esse é o nome da canção, certo?’, disse Alice, tentando se mostrar interessada.

‘Não, você não entendeu’, disse o Cavaleiro, parecendo um pouco contrariado. ‘É assim que o nome é CHAMADO. O verdadeiro nome É “O VELHO HOMEM VELHO”’

‘Então eu deveria ter dito “É assim que a CANÇÃO é chamada?”’, Alice se corrigiu.

‘Não, não deveria: isso é bem diferente! A CANÇÃO é chamada de “CAMINHOS E MEIOS”, mas é simplesmente assim que ela É CHAMADA, certo?’

‘Bem, qual É a canção então?’, disse Alice, que, a essa altura, estava completamente aturdida.

‘Eu ia chegar lá’, disse o Cavaleiro. ‘Na verdade, a música É “SENTADO NO PORTÃO”: e a melodia é minha própria invenção.’

– Lewis Carroll

Capítulo VIII, “É minha própria invenção”, Alice através do espelho

CAPÍTULO 9

Um objeto pythônico

Jamais use dois underscores na frente. Isso é irritantemente privado.¹

— Ian Bicking

Criador de pip, virtualenv, Paste e vários outros projetos

Graças ao Python data model (modelo de dados de Python), os tipos definidos pelo usuário podem se comportar tão naturalmente quanto os tipos embutidos. E isso pode ser feito sem herança, no espírito de *duck typing* (tipagem pato): basta implementar os métodos necessários para que seus objetos se comportem como esperado.

Nos capítulos anteriores, apresentamos a estrutura e o comportamento de vários tipos embutidos. Vamos agora criar classes definidas pelo usuário que se comportem como verdadeiros objetos Python.

Este capítulo começa no ponto em que o capítulo 1 terminou, mostrando como implementar diversos métodos especiais comumente vistos em objetos Python de vários tipos diferentes.

Neste capítulo, veremos como:

- Responder às funções embutidas que produzem representações alternativas de objetos (por exemplo, `repr()`, `bytes()` etc).
- Implementar um construtor alternativo como um método da classe.
- Estender a minilinguagem de formatação usada pela função embutida `format()` e pelo método `str.format()`.
- Oferecer acesso aos atributos somente para leitura.
- Fazer um objeto hashable para ser usado em conjuntos e como chaves de dict.
- Economizar memória com o uso de `__slots__`.

¹ De *Paste Style Guide* (Guia de estilo de Paste, <http://pythonpaste.org/StyleGuide.html>).

Faremos tudo isso enquanto desenvolvemos um tipo simples que representa um vetor euclidiano bidimensional.

A evolução do exemplo será interrompida para discutir dois tópicos conceituais:

- Como e quando usar os decoradores `@classmethod` e `@staticmethod`.
- Atributos privados e protegidos em Python: uso, convenções e limitações.

Vamos começar com os métodos de representação de objetos.

Representações de objetos

Toda linguagem orientada a objetos tem pelo menos uma maneira-padrão de obter uma representação em string de qualquer objeto. Python tem duas:

`repr()`

Devolve uma string que representa o objeto como o desenvolvedor quer vê-lo.

`str()`

Devolve uma string que representa o objeto como o usuário quer vê-lo.

Como você já sabe, implementamos os métodos especiais `__repr__` e `__str__` para tratar `repr()` e `str()`.

Há outros dois métodos especiais para tratar representações alternativas de objetos: `__bytes__` e `__format__`. O método `__bytes__` é análogo a `__str__`: é chamado por `bytes()` para obter o objeto representado como uma sequência de bytes. Em relação a `__format__`, tanto a função embutida `format()` quanto o método `str.format()` o chamam para obter representações de objetos em string usando códigos especiais de formatação. Abordaremos `__bytes__` no próximo exemplo, e `__format__` em seguida.



Se você estava usando Python 2, lembre-se de que em Python 3, `__repr__`, `__str__` e `__format__` sempre devolvem strings Unicode (tipo `str`). Somente `__bytes__` deve retornar uma sequência de bytes (tipo `bytes`).

Retorno da classe Vector

Para mostrar os vários métodos usados para gerar representações de objetos, usaremos uma classe `Vector2d` semelhante àquela que vimos no capítulo 1. Vamos expandi-la nesta seção e em seções mais adiante. O exemplo 9.1 mostra o comportamento básico que esperamos de uma instância de `Vector2d`.

Exemplo 9.1 – Instâncias de `Vector2d` têm várias representações

```

>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y) ❶
3.0 4.0
>>> x, y = v1 ❷
>>> x, y
(3.0, 4.0)
>>> v1 ❸
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1)) ❹
>>> v1 == v1_clone ❺
True
>>> print(v1) ❻
(3.0, 4.0)
>>> octets = bytes(v1) ❼
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1) ❽
5.0
>>> bool(v1), bool(Vector2d(0, 0)) ❾
(True, False)

```

- ❶ Os componentes de um `Vector2d` podem ser acessados diretamente como atributos (sem chamadas a métodos `getter`).
- ❷ Um `Vector2d` pode ser desempacotado em uma tupla de variáveis.
- ❸ `repr` de um `Vector2d` emula o código-fonte para construir a instância.
- ❹ Usar `eval` aqui mostra que `repr` de um `Vector2d` é uma representação fiel da chamada ao seu construtor.²
- ❺ `Vector2d` aceita comparação com `==`; isso é útil em testes.
- ❻ `print` chama `str` que, para `Vector2d`, exibe um par ordenado.
- ❼ `bytes` usa o método `__bytes__` para gerar uma representação binária.
- ❽ `abs` usa o método `__abs__` para devolver a magnitude do `Vector2d`.
- ❾ `bool` usa o método `__bool__` para devolver `False` para um `Vector2d` de magnitude zero, ou `True`, caso contrário.

`Vector2d` do exemplo 9.1 está implementado em `vector2d_v0.py` (Exemplo 9.2). O código é baseado no exemplo 1.2, mas os operadores infixos serão implementados

² Usei `eval` para clonar o objeto nesse caso, somente para enfatizar uma questão sobre `repr`; para clonar uma instância, a função `copy.copy` é mais segura e mais rápida.

no capítulo 13 – exceto `==` (que é útil para testes). A essa altura, `Vector2d` usa diversos métodos especiais para oferecer operações que um Pythonista espera de um objeto com bom design.

Exemplo 9.2 – `vector2d_v0.py`: até agora, todos os métodos são especiais

```
from array import array
import math

class Vector2d:
    typecode = 'd' ❶

    def __init__(self, x, y):
        self.x = float(x) ❷
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y)) ❸

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self) ❹

    def __str__(self):
        return str(tuple(self)) ❺

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) + ❻
                bytes(array(self.typecode, self))) ❼

    def __eq__(self, other):
        return tuple(self) == tuple(other) ❽

    def __abs__(self):
        return math.hypot(self.x, self.y) ❾

    def __bool__(self):
        return bool(abs(self)) ❿
```

- ❶ `typecode` é um atributo de classe que usaremos ao converter instâncias de `Vector2d` para/de bytes.
- ❷ Converter `x` e `y` para float em `__init__` captura erros com antecedência, o que é útil caso `Vector2d` seja chamado com argumentos inadequados.
- ❸ `__iter__` torna um `Vector2d` iterável; é isso que faz o desempacotamento funcionar (por exemplo, `x, y = my_vector`). Implementamos esse método simplesmente usando uma expressão geradora para produzir os componentes, um após o outro.³

³ Essa linha também poderia ser escrita como `yield self.x; yield self.y`. Tenho muito mais a dizer sobre o método especial `__iter__`, expressões geradoras e a palavra reservada `yield` no capítulo 14.

- ④ `__repr__` cria uma string interpolando os componentes com `{!r}` para obter sua repr; como `Vector2d` é iterável, `*self` alimenta `format` com os componentes `x` e `y`.
- ⑤ A partir de um `Vector2d` iterável, é fácil criar uma `tuple` a ser exibida como um par ordenado.
- ⑥ Para gerar bytes, convertemos o `typecode` para bytes e concatenamos...
- ⑦ ... bytes convertidos a partir de um array criado por iteração na instância.
- ⑧ Para comparar rapidamente todos os componentes, crie tuplas a partir dos operandos. Isso funciona para operandos que sejam instâncias de `Vector2d`, mas pode haver problemas. Veja o aviso a seguir.
- ⑨ A magnitude é o comprimento da hipotenusa do triângulo formado pelos componentes `x` e `y`.
- ⑩ `__bool__` usa `abs(self)` para calcular a magnitude e, em seguida, converte-a para `bool`, de modo que `0.0` torna-se `False` e um valor diferente de zero é `True`.



O método `__eq__` no exemplo 9.2 funciona para operandos `Vector2d`, mas também devolve `True` ao comparar instâncias de `Vector2d` com outros iteráveis que armazenem os mesmos valores numéricos (por exemplo, `Vector(3, 4) == [3, 4]`). Isso pode ser considerado uma funcionalidade ou um bug. Isso será discutido mais profundamente no capítulo 13, ao abordarmos a sobrecarga de operadores.

Temos um conjunto razoavelmente completo de métodos básicos, mas uma operação óbvia está faltando: recriar um `Vector2d` a partir da representação binária gerada por `bytes()`.

Um construtor alternativo

Como podemos exportar um `Vector2d` como bytes, naturalmente precisamos ter um método que importe um `Vector2d` a partir de uma sequência binária. Observando a biblioteca-padrão em busca de inspiração, vemos que `array.array` tem um método de classe chamado `.frombytes` adequado aos nossos propósitos – nós o conhecemos na seção “Arrays” na página 75. Adotamos seu nome e usamos sua funcionalidade em um método de classe para `Vector2d` em `vector2d_v1.py` (Exemplo 9.3).

Exemplo 9.3 – Parte de `vector2d_v1.py`: este trecho de código mostra somente o método de classe `frombytes`, acrescentado à definição de `Vector2d` em `vector2d_v0.py` (Exemplo 9.2)

```
@classmethod ❶  
def frombytes(cls, octets): ❷  
    typecode = chr(octets[0]) ❸  
    memv = memoryview(octets[1:]).cast(typecode) ❹  
    return cls(*memv) ❺
```

- ❶ Método de classe é modificado pelo decorador `classmethod`.
- ❷ Não há argumento `self`; em vez disso, a própria classe é passada como `cls`.
- ❸ Lê o `typecode` no primeiro byte.
- ❹ Cria uma `memoryview` a partir da sequência binária `octets` e usa o `typecode` para fazer o `cast` dos dados.⁴
- ❺ Desempacota a `memoryview` resultante do `cast`, produzindo o par de argumentos necessário ao construtor.

Como acabamos de usar um decorador `classmethod`, e ele é bem específico de Python, vamos falar um pouco sobre ele.

`classmethod` versus `staticmethod`

O decorador `classmethod` não é mencionado no tutorial de Python, nem `staticmethod`. Qualquer pessoa que tenha aprendido orientação a objetos em Java poderá se perguntar por que Python tem ambos decoradores, e não apenas um só.

Vamos começar com `classmethod`. O exemplo 9.3 mostra o seu uso: esse decorador serve para definir um método que opera na classe, e não em instâncias. `classmethod` muda a maneira como o método é chamado, de modo que ele recebe a própria classe como primeiro argumento, e não uma instância. Seu uso mais comum é em construtores alternativos, como `frombytes` no exemplo 9.3. Observe como a última linha de `frombytes` usa o argumento `cls` chamando-o para criar uma nova instância: `cls(*memv)`. Por convenção, o primeiro parâmetro de um método de classe deve se chamar `cls` (mas Python não se importa com o nome que lhe é dado).

Em comparação, o decorador `staticmethod` altera um método de modo que ele não receba um primeiro argumento especial. Essencialmente, um método estático é como uma função simples que, por acaso, reside no corpo de uma classe em vez de ser definida no nível do módulo. O exemplo 9.4 compara o funcionamento de `classmethod` e de `staticmethod`.

4 Tivemos uma rápida introdução a `memoryview` explicando seu método `.cast` na seção “Memory Views” na página 78.

Exemplo 9.4 – Comparando os comportamentos de `classmethod` e de `staticmethod`

```
>>> class Demo:
...     @classmethod
...     def klassmeth(*args):
...         return args # ❶
...     @staticmethod
...     def statmeth(*args):
...         return args # ❷
...
>>> Demo.klassmeth() # ❸
(<class '__main__.Demo'>,)
>>> Demo.klassmeth('spam')
(<class '__main__.Demo'>, 'spam')
>>> Demo.statmeth() # ❹
()
>>> Demo.statmeth('spam')
('spam',)
```

- ❶ `klassmeth` simplesmente retorna todos os argumentos posicionais.
- ❷ `statmeth` faz o mesmo.
- ❸ Não importa como você o chame, `Demo.klassmeth` recebe a classe `Demo` como primeiro argumento.
- ❹ `Demo.statmeth` se comporta exatamente como uma boa e velha função simples.



O decorador `classmethod` é obviamente útil, mas nunca vi um caso de uso que exigisse `staticmethod`. Se quiser definir uma função que não interaja com a classe, basta defini-la no módulo. Talvez a função esteja intimamente relacionada à classe, apesar de jamais entrar em contato com ela e, sendo assim, você queira tê-las próximas no código. Mesmo assim, definir a função imediatamente antes ou depois da classe no mesmo módulo será próximo o suficiente para todos os propósitos práticos.⁽⁷⁾

⁽⁷⁾ Leonardo Rochaël, um dos revisores técnicos deste livro, discorda de minha opinião desfavorável sobre `staticmethod` e recomenda o post de blog “The Definitive Guide on How to Use Static, Class or Abstract Methods in Python” (Guia definitivo para o uso de métodos estáticos, de classe ou abstratos em Python, <http://bit.ly/1FSFTW6>) de Julien Danjou para contra-argumentar. O post de Danjou é muito bom: eu recomendo. Mas não foi suficiente para me fazer mudar de ideia sobre `staticmethod`. Você deverá decidir por si mesmo.

Agora que já vimos para que serve `classmethod` (e que `staticmethod` não é muito útil), vamos voltar ao problema da representação de objetos e ver como dar suporte às saídas formatadas.

Apresentações formatadas

A função embutida `format()` e o método `str.format()` delegam a formatação propriamente dita a cada tipo chamando seu método `__format__(format_spec)`.

`format_spec` é um especificador de formatação que é:

- o segundo argumento em `format(my_obj, format_spec)` ou
- o que estiver após os dois-pontos em um campo de substituição delimitado por `{}` em uma string de formatação usada com `str.format()`.

Por exemplo:

```
>>> brl = 1/2.43 # Taxa de conversão de moeda, de BRL para USD
>>> brl
0.4115226337448559
>>> format(brl, '0.4f') # ❶
'0.4115'
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl) # ❷
'1 BRL = 0.41 USD'
```

❶ O especificador de formatação é `'0.4f'`.

❷ O especificador de formatação é `'0.2f'`. A substring `'rate'` no campo de substituição é o nome do campo. Ele não está relacionado ao especificador de formatação, mas determina qual argumento de `.format()` deve ser inserido nesse campo de substituição.

O segundo comentário numerado chama atenção para uma questão importante: uma string de formatação como `'{0.mass:5.3e}'`, na verdade, usa duas notações diferentes. `'0.mass'` à esquerda dos dois-pontos é a parte `field_name` da sintaxe do campo de substituição; `'5.3e'` após os dois-pontos é o especificador de formatação. A notação usada no especificador de formatação chama-se *Format Specification Mini-Language* (Minilinguagem de Especificação de Formatação, <http://bit.ly/1Gt4vJF>).



Se `format()` e `str.format()` forem novidade para você, a experiência em sala de aula tem mostrado que é melhor estudar a função `format()` antes, que usa somente a Minilinguagem de Especificação de Formatação (<http://bit.ly/1Gt4vJF>). Depois de pegar o jeito de `format()`, leia *Format String Syntax* (Sintaxe de formatação de string, <http://bit.ly/1Gt4vJF>) para conhecer a notação de campo de substituição `{:}` usada no método `str.format()` (incluindo as flags de conversão `!s`, `!r` e `!a`).

Alguns tipos embutidos têm seus próprios códigos de apresentação na Minilinguagem de Especificação de Formatação. Por exemplo – entre vários outros códigos –, o tipo `int` aceita `b` e `x` para saída em base 2 e em base 16, respectivamente, enquanto `float` implementa `f` para exibição de ponto fixo e `%` para exibição de porcentagem:

```
>>> format(42, 'b')
'101010'
>>> format(2/3, '.1%')
'66.7%'
```

A Miniliguagem de Especificação de Formatação é extensível, pois cada classe pode interpretar o argumento `format_spec` como quiser. Por exemplo, as classes do módulo `datetime` usam os mesmos códigos de formatação nas funções `strftime()` e em seus métodos `__format__`. Eis alguns exemplos que usam a função embutida `format()` e o método `str.format()`:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now, '%H:%M:%S')
'18:49:05'
>>> "It's now {:I:%M %p}".format(now)
"It's now 06:49 PM"
```

Se uma classe não tiver `__format__`, o método herdado de `object` devolverá `str(my_object)`. Como `Vector2d` tem um `__str__`, o código a seguir funciona:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
```

No entanto, se você passar um especificador de formatação, `object.__format__` levantará `TypeError`:

```
>>> format(v1, '.3f')
Traceback (most recent call last):
...
TypeError: non-empty format string passed to object.__format__
```

Corrigiremos isso implementando nossa própria miniliguagem de formatação. O primeiro passo será assumir que o especificador de formatação fornecido pelo usuário destina-se a formatar cada componente `float` do vetor. Este é o resultado que queremos:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

O exemplo 9.5 implementa `__format__` para gerar as apresentações que acabamos de ver.

Exemplo 9.5 – Método `Vector2d.format`, tomada #1

dentro da classe `Vector2d`

```
def __format__(self, fmt_spec=''):
    components = (format(c, fmt_spec) for c in self) # ❶
    return '({}, {})'.format(*components) # ❷
```

❶ Usa a função embutida `format` para aplicar `fmt_spec` a cada componente do vetor, criando um iterável de strings formatadas.

❷ Insere as strings formatadas na fórmula `'(x, y)'`.

Vamos agora acrescentar um código de formatação personalizado à nossa minilíngua; se o especificador de formatação terminar com `'p'`, exibiremos o vetor em coordenadas polares: $\langle r, \theta \rangle$, em que r é a magnitude e θ (teta) é o ângulo em radianos. O restante do especificador de formatação (o que quer que venha antes de `'p'`) será usado como antes.



Ao escolher a letra para o código de formatação personalizado, evitei reutilizar códigos usados por outros tipos. Na Minilíngua de Especificação de Formatação (Format Specification Mini-Language, <http://bit.ly/1Gt4vJF>), vemos que inteiros usam os códigos `'bcdxXn'`, números de ponto flutuante usam `'eEfFgGn%'` e strings usam `'s'`. Portanto escolhi `'p'` para coordenadas polares. Como cada classe interpreta esses códigos de modo independente, reutilizar uma letra de código em uma formatação personalizada para um tipo novo não é um erro, mas pode confundir os usuários.

Para gerar coordenadas polares, já temos o método `__abs__` para a magnitude, e escreveremos um método `angle` simples usando a função `math.atan2()` para obter o ângulo. Eis o código:

```
# dentro da classe Vector2d

def angle(self):
    return math.atan2(self.y, self.x)
```

Com isso, podemos melhorar nosso `__format__` para que gere coordenadas polares. Veja o exemplo 9.6.

Exemplo 9.6 – Método `Vector2d.format`, tomada #2, agora com coordenadas polares

```
def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'): ❶
        fmt_spec = fmt_spec[:-1] ❷
        coords = (abs(self), self.angle()) ❸
        outer_fmt = '<{}, {}>' ❹
```

```

else:
    coords = self ❸
    outer_fmt = '({}, {})' ❹
    components = (format(c, fmt_spec) for c in coords) ❺
    return outer_fmt.format(*components) ❻

```

- ❶ Formatação termina com 'p': use coordenadas polares.
- ❷ Remove o sufixo 'p' de `fmt_spec`.
- ❸ Cria tuple de coordenadas polares: (magnitude, angle).
- ❹ Configura a formatação externa delimitada por sinais de menor e de maior.
- ❺ Caso contrário, usa os componentes x, y de `self` para coordenadas retangulares.
- ❻ Configura a formatação externa com parênteses.
- ❼ Gera iterável com componentes como strings formatadas.
- ❽ Insere strings formatadas na formatação externa.

Com o exemplo 9.6, teremos resultados semelhantes a:

```

>>> format(Vector2d(1, 1), 'p')
'<1.4142135623730951, 0.7853981633974483>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'

```

Como mostra esta seção, não é difícil estender a minilinguagem de especificação de formatação para dar suporte a tipos definidos pelo usuário.

Vamos agora passar para um assunto que não tem a ver somente com aparências: faremos nosso `Vector2d` ser hashable para que possamos criar conjuntos de vetores ou usá-los como chaves de `dict`. Antes disso, porém, devemos tornar os vetores imutáveis. Faremos o que for necessário a seguir.

Um `Vector2d` hashable

Conforme definido, até agora, nossas instâncias de `Vector2d` não são hashable, portanto não podemos colocá-las em um `set`:

```

>>> v1 = Vector2d(3, 4)
>>> hash(v1)
Traceback (most recent call last):
...

```

```

TypeError: unhashable type: 'Vector2d'
>>> set([v1])
Traceback (most recent call last):
...
TypeError: unhashable type: 'Vector2d'

```

Para deixar um `Vector2d` hashable, devemos implementar `__hash__` (`__eq__` também é necessário, e já o temos). Também devemos deixar as instâncias de vetor imutáveis, como vimos na seção “O que é hashable?” na página 94.

Nesse momento, qualquer um pode executar `v1.x = 7`, e não há nada no código que sugira que alterar um `Vector2d` seja proibido. Este é o comportamento que queremos:

```

>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 7
Traceback (most recent call last):
...
AttributeError: can't set attribute

```

Implementaremos isso fazendo os componentes `x` e `y` serem propriedades somente para leitura no exemplo 9.7.

Exemplo 9.7 – `vector2d_v3.py`: somente as alterações necessárias para deixar `Vector2d` imutável estão mostradas aqui; veja a listagem completa no exemplo 9.9

```

class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x) ❶
        self.__y = float(y)

    @property ❷
    def x(self): ❸
        return self.__x ❹

    @property ❺
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y)) ❻

# métodos restantes vêm a seguir (omitidos na listagem do livro)

```


- ❶ Use exatamente dois underscores na frente (sem underscore no final ou com apenas um) para deixar um atributo privado.⁵
- ❷ O decorador `@property` marca o método getter de uma propriedade.
- ❸ O método getter é nomeado de acordo com a propriedade pública que ele expõe: `x`.
- ❹ Simplesmente devolve `self.__x`.
- ❺ Repete a mesma fórmula para a propriedade `y`.
- ❻ Todo método que apenas lê os componentes `x`, `y` pode permanecer inalterado, lendo as propriedades públicas com `self.x` e `self.y`, em vez de ler o atributo privado; sendo assim, essa listagem omite o restante do código da classe.



`Vector.x` e `Vector.y` são exemplos de propriedades somente para leitura. Propriedades para leitura/escrita serão discutidas no capítulo 19, quando exploraremos `@property` com mais detalhes.

Agora que nossos vetores são razoavelmente imutáveis, podemos implementar o método `__hash__`. Ele deve devolver um `int` e, idealmente, deve levar em consideração os hashes dos atributos do objeto que também são usados no método `__eq__`, pois objetos comparados como iguais devem ter o mesmo hash. A documentação do método especial `__hash__` (<https://docs.python.org/3/reference/datamodel.html>) sugere o uso do operador XOR bit a bit (^) para combinar os hashes dos componentes, portanto é isso que faremos. O código de nosso método `Vector2d.__hash__` é realmente simples, como mostra o exemplo 9.8.

Exemplo 9.8 – `vector2d_v3.py`: implementação de hash

dentro da classe `Vector2d`:

```
def __hash__(self):
    return hash(self.x) ^ hash(self.y)
```

Com a adição do método `__hash__`, agora temos vetores hashable:

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
(7, 384307168202284039)
>>> set([v1, v2])
{Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```

⁵ Não é assim que Ian Bicking faria; lembre-se da citação no início do capítulo. Os prós e contras dos atributos privados serão o assunto da próxima seção “Atributos privados e “protegidos” em Python” na página 304.



Não é estritamente necessário implementar propriedades ou proteger os atributos de instância para criar um tipo hashable. Tudo que é necessário é implementar `__hash__` e `__eq__` corretamente. Porém o valor de hash de uma instância jamais deve mudar, portanto essa é uma excelente oportunidade para falar de propriedades somente para leitura.

Se você estiver criando um tipo que tem um valor numérico escalar que faça sentido, também pode implementar os métodos `__int__` e `__float__`, chamados pelos construtores `int()` e `float()` – usados para conversão de tipo em alguns contextos. Há também um método `__complex__` para dar suporte ao construtor embutido `complex()`. Talvez `Vector2d` deva oferecer `__complex__`, mas deixarei isso como exercício para você.

Estivemos trabalhando em `Vector2d` há um tempo, mostrando apenas trechos de código; o exemplo 9.9 é uma listagem completa e consolidada de `vector2d_v3.py`, incluindo todos os doctests que usei para desenvolvê-lo.

Exemplo 9.9 – `vector2d_v3.py`: completo

```
'''
```

Uma classe de vetor bidimensional

```
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)
3.0 4.0
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector2d(0, 0))
(True, False)
```

Teste do método de classe ``.frombytes()``:

```
>>> v1_clone = Vector2d.frombytes(bytes(v1))
```

```
>>> v1_clone
Vector2d(3.0, 4.0)
>>> v1 == v1_clone
True
```

Testes de ``format()`` com coordenadas cartesianas:

```
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

Testes do método ``angle``:

```
>>> Vector2d(0, 0).angle()
0.0
>>> Vector2d(1, 0).angle()
0.0
>>> epsilon = 10**-8
>>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
True
>>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
True
```

Testes de ``format()`` com coordenadas polares:

```
>>> format(Vector2d(1, 1), 'p') # doctest:+ELLIPSIS
'<1.414213..., 0.785398...>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

Testes das propriedades ``x`` e ``y`` somente para leitura:

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 123
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

Testes de hashing:

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
(7, 384307168202284039)
>>> len(set([v1, v2]))
2
```

```
"""
```

```
from array import array
```

```
import math
```

```
class Vector2d:
```

```
    typecode = 'd'
```

```
    def __init__(self, x, y):
```

```
        self.__x = float(x)
```

```
        self.__y = float(y)
```

```
    @property
```

```
    def x(self):
```

```
        return self.__x
```

```
    @property
```

```
    def y(self):
```

```
        return self.__y
```

```
    def __iter__(self):
```

```
        return (i for i in (self.x, self.y))
```

```
    def __repr__(self):
```

```
        class_name = type(self).__name__
```

```
        return '{}({!r}, {!r})'.format(class_name, *self)
```

```
    def __str__(self):
```

```
        return str(tuple(self))
```

```
    def __bytes__(self):
```

```
        return (bytes([ord(self.typecode)]) +
```

```
                bytes(array(self.typecode, self)))
```

```
    def __eq__(self, other):
```

```
        return tuple(self) == tuple(other)
```

```
    def __hash__(self):
```

```
        return hash(self.x) ^ hash(self.y)
```

```
    def __abs__(self):
```

```
        return math.hypot(self.x, self.y)
```

```

def __bool__(self):
    return bool(abs(self))

def angle(self):
    return math.atan2(self.y, self.x)

def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'):
        fmt_spec = fmt_spec[:-1]
        coords = (abs(self), self.angle())
        outer_fmt = '<{}, {}>'
    else:
        coords = self
        outer_fmt = '({}, {})'
    components = (format(c, fmt_spec) for c in coords)
    return outer_fmt.format(*components)

@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(*memv)

```

Para recapitular, nesta seção e nas seções anteriores, vimos alguns métodos especiais essenciais que você pode implementar para ter um objeto completo. É claro que implementar todos esses métodos sem que sua aplicação precise deles não é uma boa ideia. Para os clientes que pagam pelo desenvolvimento, não importa se seus objetos são ou não “pythônicos”.

Conforme implementado no exemplo 9.9, `Vector2d` é um exemplo didático, com uma longa lista de métodos especiais relacionados à representação do objeto, e não um template para todas as classes que você vai desenvolver.

Na próxima seção, deixaremos `Vector2d` um pouco de lado para discutir o design e as desvantagens do mecanismo de atributos privados em Python – o prefixo com underscore duplo em `self.__x`.

Atributos privados e “protegidos” em Python

Em Python, não há como criar variáveis privadas como é feito com o modificador `private` em Java. O que temos em Python é um mecanismo simples para evitar que um atributo “privado” seja sobrescrito acidentalmente em uma subclasse.

Considere este cenário: alguém escreveu uma classe chamada `Dog` que usa um atributo de instância `mood` internamente, sem expô-lo. Você precisa criar a subclasse `Beagle` de

Dog. Se criar seu próprio atributo de instância `mood`, sem ter ciência do conflito de nomes, você afetará o atributo `mood` usado pelos métodos herdados de `Dog`. Será bem difícil para depurar.

Para evitar isso, se você nomear um atributo de instância no formato `__mood` (dois underscores na frente e nenhum, ou no máximo um underscore no final), Python armazenará o nome no `__dict__` da instância prefixado com um underscore na frente e o nome da classe, de modo que, na classe `Dog`, `__mood` será `_Dog__mood`, e em `Beagle` será `_Beagle__mood`. Esse recurso da linguagem tem o nome simpático de *desfiguração de nomes* (name mangling).

O exemplo 9.10 mostra o resultado na classe `Vector2d` do exemplo 9.7.

Exemplo 9.10 – Nomes de atributos privados são “desfigurados” com `_` e o nome da classe como prefixos

```
>>> v1 = Vector2d(3, 4)
>>> v1.__dict__
{'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
>>> v1._Vector2d__x
3.0
```

A desfiguração de nomes tem a ver com proteção, e não com segurança: foi concebida para evitar acessos acidentais, e não contra ações maliciosas intencionais (a figura 9.1 mostra outro dispositivo de proteção).

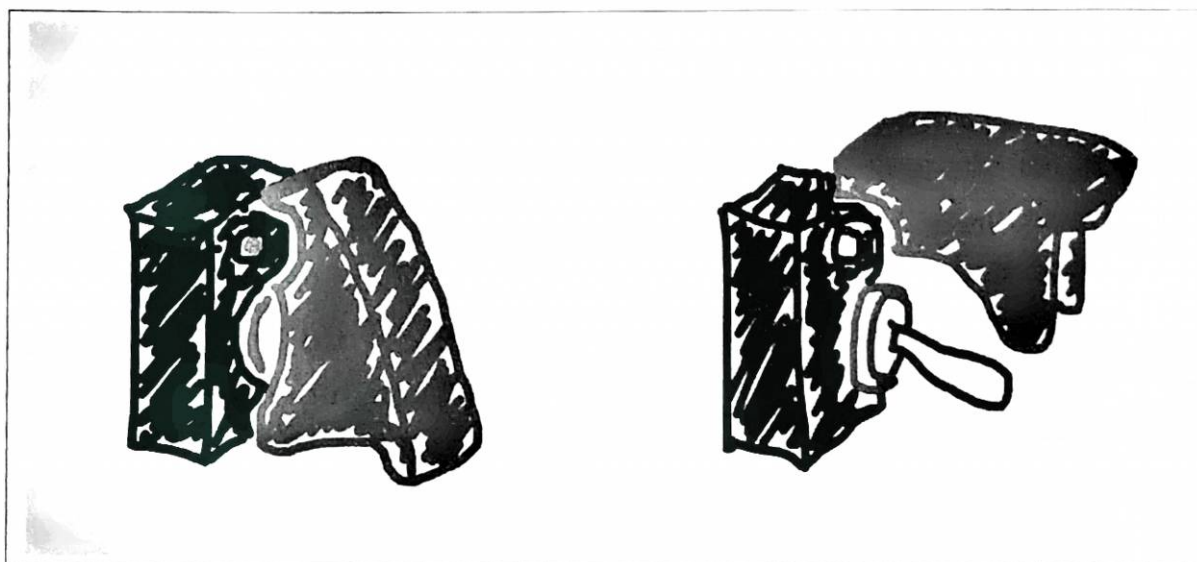


Figura 9.1 – Uma capa sobre um interruptor é um dispositivo de proteção, e não de segurança: ela evita uma ativação acidental, mas não impede um uso malicioso.

Qualquer pessoa que saiba como os nomes privados são desfigurados pode ler o atributo privado diretamente, como mostra a última linha do exemplo 9.10 – na verdade, isso é útil para depuração e serialização. Qualquer um também pode atribuir um valor diretamente a um componente privado de um `Vector2d` simplesmente escrevendo

`v1._Vector__x = 7`. Mas se você fizer isso em código de produção, não poderá reclamar caso algo dê errado.

Nem todo pythonista ama a funcionalidade de desfiguração de nomes ou a aparência torta dos nomes escritos como `self.__x`. Alguns preferem evitar essa sintaxe e usam somente um underscore como prefixo para “proteger” atributos por convenção (por exemplo, `self._x`). Os críticos da desfiguração automática com underscores duplos sugerem que preocupações com a alteração acidental de atributos devem ser tratadas por convenções de nomenclatura. A seguir, apresentamos a citação completa do produtivo Ian Bicking parcialmente exibida no início deste capítulo:

Jamais use dois underscores na frente. Isso é irritantemente privado. Se conflitos de nomes for uma preocupação, use desfiguração explícita de nomes em seu lugar (por exemplo, `_MyThing_blahblah`). É essencialmente igual a usar underscores duplos, exceto por ser transparente, enquanto underscores duplos são obscuros.⁶

O prefixo com apenas um underscore não tem significado especial para o interpretador Python quando usado em nomes de atributos, mas, entre programadores Python, não acessar esses atributos de fora da classe é uma convenção bem respeitada.⁷ É fácil respeitar a privacidade de um objeto que marque seus atributos com um único `_`, assim como é fácil respeitar a convenção segundo a qual as variáveis em `ALL_CAPS` devem ser tratadas como constantes.

Atributos com apenas um `_` como prefixo são chamados de “protegidos” em alguns pontos da documentação de Python.⁸ A prática de “proteger” atributos por convenção com o formato `self._x` é amplamente disseminada, mas chamar isso de atributo “protegido” não é tão comum. Algumas pessoas até os chamam de atributos “privados”.

Para concluir: os componentes de `Vector2d` são “privados” e nossas instâncias de `Vector2d` são “imutáveis” – assim, com aspas – porque não há maneira de torná-los realmente privados e imutáveis.⁹

Agora vamos voltar à nossa classe `Vector2d`. Nesta última seção abordaremos um atributo especial (e não um método) que afeta o armazenamento interno de um objeto, possivelmente com enorme impacto no uso de memória, mas pouco efeito em sua interface pública: `__slots__`.

6 De *Paste Style Guide* (Guia de estilo de Paste, <http://pythonpaste.org/StyleGuide.html>).

7 Em módulos, um único `_` na frente de um nome de nível mais alto tem efeito: se você escrever `from mymod import *`, os nomes com um prefixo `_` não serão importados de `mymod`. Contudo você pode continuar escrevendo `from mymod import _privatefunc`. Isso é explicado no Tutorial de Python na seção 6.1 *More on Modules* (Mais sobre módulos, <http://bit.ly/1Gt95rp>).

8 Um exemplo está na documentação do módulo `gettext` (<http://bit.ly/1Gt9cDg>).

9 Se essa situação deixa você deprimido e o faz desejar que Python fosse mais parecido com Java quanto a esse aspecto, não leia minha discussão sobre a robustez relativa do modificador `private` em Java na seção “Ponto de vista” na página 316.

Economizando espaço com o atributo de classe `__slots__`

Por padrão, Python armazena atributos de instância em um `dict` chamado `__dict__` em cada instância. Como vimos na seção “Consequências práticas de como os dicionários funcionam” na página 121, os dicionários têm um overhead significativo de memória por causa da tabela hash subjacente usada para possibilitar acessos rápidos. Se você estiver lidando com milhões de instâncias com poucos atributos, o atributo de classe `__slots__` é capaz de economizar bastante memória ao permitir que o interpretador armazene os atributos de instância em um `tuple` em vez de usar um `dict`.



Um atributo `__slots__` herdado de uma superclasse não tem nenhum efeito. Python leva em consideração somente os atributos `__slots__` definidos em cada classe individualmente.

Para definir `__slots__`, crie um atributo de classe com esse nome e atribua-lhe um iterável de `str` com identificadores para os atributos de instância. Gosto de usar um `tuple` para isso, pois ele transmite a mensagem de que a definição de `__slots__` não pode mudar. Veja o exemplo 9.11.

Exemplo 9.11 – `vector2d_v3_slots.py`: o atributo `slots` é o único acréscimo a `Vector2d`

```
class Vector2d:
    __slots__ = ('__x', '__y')
    typecode = 'd'

    # métodos vêm a seguir (omitidos na listagem do livro)
```

Ao definir `__slots__` na classe, você estará dizendo o seguinte ao interpretador: “Esses são todos os atributos de instância dessa classe.” Python então os armazena em uma estrutura parecida com uma tupla em cada instância, evitando o overhead de memória do `__dict__` por instância. Isso pode fazer uma enorme diferença no uso de memória caso você tenha milhões de instâncias ativas ao mesmo tempo.



Se estiver lidando com milhões de objetos com dados numéricos, você realmente deve usar arrays NumPy (veja a seção “NumPy e SciPy” na página 80), que não só são eficientes quanto ao uso de memória como também têm funções extremamente otimizadas para processamentos numéricos, muitas das quais atuam em todo o array de uma só vez. Projetei a classe `Vector2d` somente para oferecer um contexto para a discussão dos métodos especiais, pois, sempre que posso, procuro evitar exemplos vagos com `foo` e `bar`.

O exemplo 9.12 mostra duas execuções de um script que simplesmente criam uma list usando uma list comprehension, com 10 milhões de instâncias de `Vector2d`. O

script *mem_test.py* aceita o nome de um módulo com uma variante da classe `Vector2d` como argumento de linha de comando. Na primeira execução, usei `vector2d_v3.Vector2d` (do exemplo 9.7); na segunda execução, a versão de `vector2d_v3_slots.Vector2d` com `__slots__` foi usada.

Exemplo 9.12 – *mem_test.py* cria 10 milhões de instâncias de `Vector2d` usando a classe definida no módulo especificado (por exemplo, *vector2d_v3.py*)

```
$ time python3 mem_test.py vector2d_v3.py
Selected Vector2d type: vector2d_v3.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      5,623,808
  Final RAM usage:  1,558,482,944

real 0m16.721s
user 0m15.568s
sys 0m1.149s
$ time python3 mem_test.py vector2d_v3_slots.py
Selected Vector2d type: vector2d_v3_slots.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      5,718,016
  Final RAM usage:  655,466,496

real 0m13.605s
user 0m13.163s
sys 0m0.434s
```

Como mostra o exemplo 9.12, o uso de RAM pelo script aumenta para 1.5 GB quando um `__dict__` por instância é usado em cada uma de 10 milhões de instâncias de `Vector2d`, porém esse valor é reduzido para 655 MB quando `Vector2d` tem um atributo `__slots__`. A versão com `__slots__` também é mais rápida. O script *mem_test.py* nesse teste basicamente lida com a carga de um módulo, a verificação do uso de memória e a formatação dos resultados. O código não é realmente relevante nesse caso e, sendo assim, está no Apêndice A, no exemplo A.4.



Quando `__slots__` é especificado em uma classe, suas instâncias não podem ter outros atributos além daqueles nomeados em `__slots__`. Esse é realmente um efeito colateral, e não o motivo para a existência de `__slots__`. Usar `__slots__` somente para evitar que os usuários de sua classe criem novos atributos nas instâncias, caso eles queiram, não é considerado uma boa solução. `__slots__` deve ser usado para otimização, e não para restringir os programadores.

Contudo é possível “economizar memória e consumi-la também”: se você adicionar o nome `'__dict__'` à lista `__slots__`, suas instâncias manterão os atributos nomeados em

`__slots__` na tupla por instância, mas também oferecerão suporte a atributos criados dinamicamente, que serão armazenados no `__dict__` usual. É claro que ter '`__dict__`' em `__slots__` pode anular totalmente o seu propósito, dependendo do número de atributos estáticos e dinâmicos em cada instância e de como eles são usados. Uma otimização descuidada é pior que uma otimização prematura.

Há outro atributo especial por instância que talvez você queira manter: o atributo `__weakref__` é necessário a um objeto para dar suporte a referências fracas (discutido na seção “Referências fracas” na página 276). Por padrão, esse atributo está presente em instâncias de classes definidas pelo usuário. Entretanto, se a classe definir `__slots__` e você quiser que as instâncias sejam alvos de referências fracas, será necessário incluir '`__weakref__`' entre os atributos nomeados em `__slots__`.

Em suma, `__slots__` deve ser usado com algumas ressalvas e não se deve abusar dele somente para limitar quais atributos podem ser definidos pelos usuários. Ele é mais útil quando trabalhamos com dados tabulares como registros de banco de dados, em que o esquema é fixo por definição e os conjuntos de dados podem ser bem grandes. No entanto, se você fizer esse tipo de trabalho com frequência, dê uma olhada não só em NumPy (<http://www.numpy.org/>) como também na biblioteca de análise de dados Pandas (<http://pandas.pydata.org/>), que pode manipular dados não numéricos e importar/exportar para vários formatos de dados tabulares diferentes.

Os problemas com `__slots__`

Para resumir, `__slots__` pode possibilitar economias significativas de memória se for usado de forma apropriada, mas há algumas ressalvas:

- Você deve se lembrar de declarar `__slots__` novamente em cada subclasse, pois o atributo herdado é ignorado pelo interpretador.
- As instâncias poderão ter somente os atributos listados em `__slots__`, a menos que você inclua '`__dict__`' em `__slots__` (mas fazer isso poderá anular a economia de memória).
- As instâncias não poderão ser alvos de referências fracas, a menos que você se lembre de incluir '`__weakref__`' em `__slots__`.

Se o seu programa não trata milhões de instâncias, provavelmente não valerá a pena dar-se o trabalho de criar uma classe, de certo modo, incomum e complicada, cujas instâncias não aceitam atributos dinâmicos ou referências fracas. Como qualquer otimização, `__slots__` deve ser utilizado somente se uma necessidade justificar seu uso no momento e quando seus benefícios forem comprovados por meio de uma cuidadosa análise do perfil de execução (profiling).

O último tópico deste capítulo tem a ver com sobrescrita de um atributo de classe em instâncias e em subclasses.

Sobrescrita de atributos de classe

Um recurso notável de Python é o uso de atributos de classe como valores default para atributos de instância. Em `Vector2d`, há um atributo de classe chamado `typecode`. Ele é usado duas vezes no método `__bytes__`, mas nós o acessamos na instância, como `self.typecode`, de propósito. Como as instâncias de `Vector2d` são criadas sem um atributo `typecode` próprio, `self.typecode` obterá o atributo de classe `Vector2d.typecode` por padrão.

No entanto, se você escrever em um atributo de instância que não exista, um novo atributo de instância será criado – por exemplo, o atributo de instância `typecode` – e o atributo de classe de mesmo nome permanecerá inalterado. Porém, a partir daí, sempre que o código que trata essa instância ler `self.typecode`, o `typecode` da instância será recuperado, ocultando o atributo de classe de mesmo nome. Isso abre a possibilidade de personalizar uma instância individual com um `typecode` diferente.

O default de `Vector2d.typecode` é `'d'`, o que significa que cada componente do vetor será representado como um número de ponto flutuante de dupla precisão com 8 bytes ao ser exportado para bytes. Se definirmos o `typecode` de uma instância de `Vector2d` com `'f'` antes de exportar os dados, cada componente será exportado como um número de ponto flutuante de precisão simples de 4 bytes. O exemplo 9.13 mostra isso.



Estamos discutindo a adição de um atributo de instância personalizado e, sendo assim, o exemplo 9.13 usa a implementação de `Vector2d` sem `__slots__`, conforme mostrado no exemplo 9.9.

Exemplo 9.13 – Personalizando uma instância definindo o atributo `typecode` que havia sido previamente herdado da classe

```
>>> from vector2d_v3 import Vector2d
>>> v1 = Vector2d(1.1, 2.2)
>>> dumpd = bytes(v1)
>>> dumpd
b'd\x9a\x99\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x99\x01@'
>>> len(dumpd) # ❶
17
>>> v1.typecode = 'f' # ❷
>>> dumpf = bytes(v1)
>>> dumpf
```

```
b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
>>> len(dumpf) # ❸
9
>>> Vector2d.typecode # ❹
'd'
```

❶ Representação default em bytes tem 17 bytes.

❷ Define typecode com 'f' na instância v1.

❸ Agora o dump de bytes tem 9 bytes.

❹ Vector2d.typecode permanece inalterado; somente a instância v1 usa o typecode 'f'.

Agora deve estar claro por que a exportação de bytes de um Vector2d é prefixada pelo typecode: queríamos oferecer suporte a diferentes formatos de exportação.

Se quiser alterar um atributo de classe, você deve defini-lo diretamente na classe, e não por meio de uma instância. Você poderia alterar o typecode default para todas as instâncias (que não tenham seu próprio typecode) assim:

```
>>> Vector2d.typecode = 'f'
```

No entanto há uma maneira idiomática em Python de conseguir um efeito permanente e ser mais explícito quanto à mudança. Como os atributos de classe são públicos, eles são herdados pelas subclasses, portanto criar uma subclasse apenas para personalizar um atributo de dados de classe é uma prática comum. As views baseadas em classes de Django usam bastante essa técnica. O exemplo 9.14 mostra como isso é feito.

Exemplo 9.14 – ShortVector2d é uma subclasse de Vector2d, que apenas sobrescreve o typecode default

```
>>> from vector2d_v3 import Vector2d
>>> class ShortVector2d(Vector2d): # ❶
...     typecode = 'f'
...
>>> sv = ShortVector2d(1/11, 1/27) # ❷
>>> sv
ShortVector2d(0.09090909090909091, 0.037037037037037035) # ❸
>>> len(bytes(sv)) # ❹
9
```

❶ Cria ShortVector2d como uma subclasse de Vector2d somente para sobrescrever o atributo de classe typecode.

❷ Cria a instância sv de ShortVector2d para demonstração.

❸ Inspecciona repr de sv.

❹ Verifica se o tamanho dos bytes exportados é 9, e não 17, como antes.

Esse exemplo também explica por que não deixei `class_name` fixo no código em `Vector2d.__repr__`, mas o obtive de `type(self).__name__`, desta maneira:

```
# dentro da classe Vector2d:

def __repr__(self):
    class_name = type(self).__name__
    return '{}({!r}, {!r})'.format(class_name, *self)
```

Se eu tivesse deixado `class_name` fixo no código, as subclasses de `Vector2d`, como `ShortVector2d`, teriam de sobrescrever `__repr__` somente para alterar `class_name`. Ao ler o nome do `type` da instância, deixei `__repr__` mais seguro para ser herdado.

Com isso, terminamos nossa discussão sobre a implementação de uma classe simples que aproveita o modelo de dados a fim de ter uma boa integração com o restante de Python – oferecendo diferentes representações de objetos, implementando um código de formatação personalizado, expondo atributos somente para leitura e dando suporte a `hash()` para integração com conjuntos e mapeamentos.

Resumo do capítulo

O objetivo deste capítulo foi mostrar o uso de métodos especiais e de convenções na construção de uma classe pythônica bem comportada.

vector2d_v3.py (Exemplo 9.9) é mais pythônico que *vector2d_v0.py* (Exemplo 9.2)? A classe `Vector2d` em *vector2d_v3.py* certamente exhibe mais recursos de Python. Porém, se a primeira ou a última implementação de `Vector2d` é mais idiomática depende do contexto em que ela será usada. O Zen do Python de Tim Peter afirma que:

Simples é melhor que complexo.

Um objeto pythônico deve ser tão simples quanto permitirem os requisitos – não deve ser um desfile de recursos da linguagem.

Porém meu objetivo ao expandir o código de `Vector2d` foi fornecer um contexto para discutir métodos especiais de Python e convenções de codificação. Se você observar novamente a tabela 1.1, verá que as várias listagens neste capítulo mostraram:

- todos os métodos para representação em string/bytes: `__repr__`, `__str__`, `__format__` e `__bytes__`.
- vários métodos para converter um objeto em um número: `__abs__`, `__bool__`, `__hash__`.
- o operador `__eq__` para testar conversão de bytes e possibilitar hashing (juntamente com `__hash__`).

Ao dar suporte à conversão para bytes, implementamos também um construtor alternativo, `Vector2d.frombytes()`, que forneceu o contexto para discutir os decoradores `@classmethod` (muito prático) e `@staticmethod` (não tão útil; uma função no nível do módulo é mais simples). O método `frombytes` foi inspirado em seu homônimo na classe `array.array`.

Vimos que a Minilinguagem de Especificação de Formatação (Format Specification Mini-Language, <https://docs.python.org/3/library/string.html#formatspec>) é extensível implementando um método `__format__` que faz um parsing mínimo em `format_spec` fornecido à função embutida `format(obj, format_spec)` ou em campos de substituição `'{:«format_spec»}'` em strings usadas com o método `str.format`.

Na preparação para deixar as instâncias de `Vector2d` hashable, fizemos um esforço para torná-las imutáveis, pelo menos evitando alterações acidentais ao codar os atributos `x` e `y` como privados e expondo-os como propriedades somente para leitura. Em seguida, implementamos `__hash__` usando a técnica recomendada de “xor” nos hashes dos atributos de instância.

Em seguida, discutimos as economias de memória e as ressalvas ligadas à declaração de um atributo `__slots__` em `Vector2d`. A forma de usar `__slots__`, de certo modo, tem suas artimanhas; isso realmente faz sentido somente quando estamos tratando uma quantidade muito grande de instâncias – pense em milhões de instâncias, e não apenas em milhares.

O último tópico abordado foi a sobrescrita de um atributo de classe acessado por meio das instâncias (por exemplo, `self.typecode`). Fizemos isso inicialmente criando um atributo de instância e, em seguida, criando uma subclasse e sobrescrevendo o atributo no nível da classe.

Ao longo do capítulo, mencionei como as opções de design nos exemplos foram bem fundamentadas estudando a API de objetos-padrão em Python. Se este capítulo pudesse ser sintetizado em uma frase, seria esta:

Para criar objetos pythônicos, observe como os objetos Python de verdade se comportam.

— Antigo provérbio chinês

Leituras complementares

Este capítulo abordou vários métodos especiais do modelo de dados; sendo assim, as principais referências naturalmente são as mesmas indicadas no capítulo 1, que ofereceu uma visão geral sobre o mesmo assunto. Por conveniência, repito aqui as quatro recomendações anteriores e acrescentarei outras:

Capítulo “Data Model” (Modelo de dados, <http://bit.ly/1GsZwss>) de *The Python Language Reference* (Guia de referência à linguagem Python)

A maioria dos métodos que usamos neste capítulo está documentada na seção “3.3.1. Basic customization” (Personalização básica, <http://bit.ly/1Vma6b2>).

Python in a Nutshell, 2ª edição, de Alex Martelli (<http://shop.oreilly.com/product/9780596100469.do>)

Excelente discussão sobre o modelo de dados, apesar de abordar apenas Python 2.5 (na segunda edição). Os conceitos fundamentais são todos iguais, e a maior parte das APIs do Data Model não mudou desde Python 2.2, quando tipos embutidos e classes definidas pelo usuário se tornaram mais compatíveis.

Python Cookbook, 3ª edição, de David Beazley e Brian K. Jones (O’Reilly, <http://shop.oreilly.com/product/0636920027072.do>)¹⁰

Práticas bem modernas de implementação mostradas por meio de receitas. O capítulo 8, “Classes and Objects” (Classes e Objetos), particularmente tem diversas soluções relacionadas às discussões deste capítulo.

Python Essential Reference, 4ª edição, de David Beazley

Discute o modelo de dados em detalhes no contexto de Python 2.6 e Python 3.

Neste capítulo, abordamos todos os métodos especiais relacionados à representação de objetos, exceto `__index__`. Ele é usado para coerção de um objeto para um índice inteiro no contexto específico de fatiamento de sequências e foi criado para atender a uma necessidade de NumPy. Na prática, você e eu provavelmente não precisaremos implementar `__index__`, a menos que a gente decida criar um novo tipo de dado numérico e queira usá-lo como argumento de `__getitem__`. Se estiver curioso a respeito desse método, *What’s New in Python 2.5* (O que há de novo em Python 2.5, <https://docs.python.org/2.5/whatsnew/pep-357.html>) tem uma pequena explicação, e a PEP 357 – *Allowing Any Object to be Used for Slicing* (Permitindo que qualquer objeto seja usado para fatiamento, <https://www.python.org/dev/peps/pep-0357/>) detalha por que `__index__` é necessário do ponto de vista de Travis Oliphant, que implementou uma extensão em C e é o principal autor de NumPy.

Uma das primeiras vezes em que se percebeu a necessidade de ter representações distintas em string para os objetos foi em Smalltalk. O artigo “*How to Display an Object as a String: printString and displayString*” (Como exibir um objeto como uma string: `printString` e `displayString`, <http://bit.ly/1IKX6t>) de Bobby Woolf em 1996 discute a implementação dos métodos `printString` e `displayString` naquela linguagem. Desse artigo, emprestei as descrições expressivas “como o desenvolvedor quer vê-lo” e “como

¹⁰ N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

o usuário quer vê-lo” na definição de `repr()` e de `str()` na seção “Representações de objetos” na página 289.

Ponto de vista

Propriedades ajudam a reduzir custos iniciais

Nas primeiras versões de `Vector2d`, os atributos `x` e `y` eram públicos, como são todos os atributos de instância e de classe em Python por padrão. Naturalmente, os usuários de vetores devem poder acessar seus componentes. Embora nossos vetores sejam iteráveis e possam ser desempacotados em um par de variáveis, também é desejável que possamos escrever `my_vector.x` e `my_vector.y` para obter cada componente.

Quando sentimos a necessidade de evitar atualizações acidentais nos atributos `x` e `y`, implementamos propriedades, mas nada mudou em outros pontos do código e na interface pública de `Vector2d`, conforme atestaram os doctests. Ainda podemos acessar `my_vector.x` e `my_vector.y`.

Isso mostra que sempre podemos iniciar nossas classes da maneira mais simples possível, com atributos públicos, pois quando (ou se), mais tarde, precisarmos impor mais controle com getters e setters, eles poderão ser implementados com propriedades, sem alterar códigos que já interajam com nossos objetos por meio de nomes (por exemplo, `x` e `y`) que, inicialmente, eram atributos públicos simples.

Essa abordagem é oposta àquela incentivada pela linguagem Java: um programador Java não pode começar com atributos públicos simples e, somente depois, se for necessário, implementar propriedades, pois elas não existem na linguagem. Sendo assim, escrever getters e setters é a norma em Java – mesmo quando esses métodos não façam nada útil – porque a API não pode evoluir de atributos públicos simples para getters e setters, sem quebrar todo código que use esses atributos.

Além do mais, como nosso revisor técnico Alex Martelli destaca, digitar chamadas a getter/setter por todo lado é tolo. Você é obrigado a escrever algo como:

```
...  
>>> my_object.set_foo(my_object.get_foo() + 1)
```

```
...  
  
para fazer apenas isto:
```

```
...  
>>> my_object.foo += 1
```

```
...
```

Ward Cunningham, inventor do wiki e pioneiro de Extreme Programming, recomenda perguntar: “Qual é a solução mais simples que possivelmente poderia funcionar?”. A ideia é focar no objetivo.¹¹ Implementar setters e getters antecipadamente é uma distração em relação à meta. Em Python, podemos simplesmente usar atributos públicos, sabendo que podemos alterá-los depois para propriedades, se for necessário.

Proteção versus segurança em atributos privados

Perl não tem paixão por privacidade imposta. Ela prefere que você fique fora de sua sala de estar porque não foi convidado, e não porque ela tem uma arma.

— Larry Wall
Criador de Perl

Python e Perl são opostos em vários aspectos, mas Larry e Guido parecem concordar quanto à privacidade de objetos.

Tendo ensinado Python a muitos programadores de Java ao longo dos anos, descobri que muitos deles depositam muita fé nas garantias de privacidade oferecidas por Java. O fato é que os modificadores `private` e `protected` de Java normalmente oferecem proteção somente contra acidentes. Eles podem garantir a segurança contra intenções maliciosas somente se a aplicação for instalada com um security manager (gerenciador de segurança), e isso raramente acontece na prática, mesmo em ambientes corporativos.

Para provar o meu ponto de vista, gostaria de mostrar a classe Java a seguir (Exemplo 9.15).

Exemplo 9.15 – Confidential.java: uma classe Java com um campo privado chamado `secret`

```
public class Confidential {  
    private String secret = "";  
    public Confidential(String text) {  
        secret = text.toUpperCase();  
    }  
}
```

No exemplo 9.15, armazenei `text` no campo `secret` após tê-lo convertido para letras maiúsculas somente para deixar evidente que o que quer que esteja nesse campo terá somente letras maiúsculas.

¹¹ Veja “Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V” (A solução mais simples que possivelmente poderia funcionar: uma conversa com Ward Cunningham, Parte V, <http://www.artima.com/intv/simplest3.html>).

A demonstração propriamente dita consiste em executar *expose.py* com Jython. Esse script usa introspecção (“reflexão” no jargão de Java) para obter o valor de um campo privado. O código está no exemplo 9.16.

Exemplo 9.16 – *expose.py*: código Jython para ler o conteúdo de um campo privado em outra classe

```
import Confidential

message = Confidential('top secret text')
secret_field = Confidential.getDeclaredField('secret')
secret_field.setAccessible(True) # quebramos o cadeado!
print 'message.secret = ', secret_field.get(message)
```

Se executar o exemplo 9.16, verá o seguinte:

```
$ jython expose.py
message.secret = TOP SECRET TEXT
```

A string 'TOP SECRET TEXT' foi lida do campo privado *secret* da classe *Confidential*.

Não há nenhuma magia negra aqui: *expose.py* usa a API de reflexão de Java para obter uma referência ao campo privado chamado 'secret' e, em seguida, chama 'secret_field.setAccessible(True)' para que ele possa ser lido. O mesmo pode ser feito com código Java, é claro (mas é preciso mais que o triplo de linhas para isso; veja o arquivo *Expose.java* no repositório de código de *Python fluente* [<https://github.com/fluentpython/example-code>]).

A chamada crucial *.setAccessible(True)* falhará somente se o script Jython ou o programa principal Java (por exemplo, *Expose.class*) estiver executando sob a supervisão de um *SecurityManager* (<http://bit.ly/IIIdqd>). Porém, no mundo real, as aplicações Java raramente são instaladas com um *SecurityManager* – exceto as applets Java (lembra delas?).

O ponto que quero destacar é que, também em Java, os modificadores de controle de acesso têm mais a ver com proteção do que com segurança, pelo menos na prática. Portanto relaxe e desfrute do poder que Python oferece a você. Use-o com responsabilidade.

CAPÍTULO 10

Hackeando e fatiando sequências

Não verifique se *é* um pato; verifique se faz *quack* como um pato, *anda* como um pato etc., etc., de acordo com o subconjunto exato de comportamentos de pato de que você precisa para jogar com a linguagem. (comp.lang.python, 26 de julho de 2000)

—Alex Martelli

Neste capítulo, criaremos uma classe para representar uma classe **Vector** multidimensional – um passo significativo para além do **Vector2d** bidimensional do capítulo 9. **Vector** se comportará como uma sequência-padrão simples e imutável em Python. Seus elementos serão números de ponto flutuante **e**, no final deste capítulo, essa classe oferecerá suporte para:

- protocolo básico de sequência: `__len__` e `__getitem__`;
- representação segura de instâncias com vários itens;
- fatiamento correto, produzindo novas instâncias de **Vector**;
- hashing agregado, levando em conta os valores de todos os elementos contidos;
- extensão personalizada à linguagem de formatação.

Também implementaremos acesso dinâmico a atributos com `__getattr__` como uma maneira de substituir as propriedades somente para leitura que usamos em **Vector2d** – embora isso não seja típico de sequências.

A apresentação intensiva em código será interrompida por uma discussão conceitual da ideia de protocolos como interfaces informais. Discutiremos a relação entre protocolos e *duck typing* (tipagem pato) e as implicações práticas de criar seus próprios tipos. Vamos começar.

Aplicações de vetores além de três dimensões

Quem precisa de um vetor com mil dimensões? Dica: não os artistas 3D! No entanto vetores n -dimensionais (com valores elevados de n) são amplamente usados na recuperação de informações, em que consultas a documentos e textos são representadas como vetores, com uma dimensão por palavra. Isso se chama modelo de espaço vetorial (http://en.wikipedia.org/wiki/Vector_space_model). Nesse modelo, uma métrica de relevância fundamental é a similaridade de cosseno (isto é, o cosseno do ângulo entre o vetor que representa a consulta e o vetor que representa um documento). À medida que o ângulo diminui, o cosseno aproxima-se do valor máximo, que é 1, e o mesmo ocorre com a relevância do documento para a consulta.

Apesar do que foi dito, a classe `Vector` deste capítulo é um exemplo didático, e não usaremos muita matemática aqui. Nosso objetivo é apenas mostrar alguns métodos especiais de Python no contexto de um tipo de sequência.

NumPy e SciPy são as ferramentas de que você precisa para operações matemáticas com vetores no mundo real. O pacote `gensim` (<https://pypi.python.org/pypi/gensim>) de Radim Rehurek no PyPi implementa modelagem de espaço vetorial para processamento de linguagem natural e recuperação de informações usando NumPy e SciPy.

Vector: um tipo de sequência definido pelo usuário

Nossa estratégia para implementar `Vector` será usar composição, e não herança. Armazenaremos os componentes em um array de números de ponto flutuante e implementaremos os métodos necessários ao nosso `Vector` para que ele se comporte como uma sequência simples imutável.

Contudo, antes de implementar os métodos de sequência, vamos garantir que temos uma implementação básica de `Vector` compatível com nossa classe `Vector2d` anterior – exceto nos pontos em que essa compatibilidade não fizer sentido.

Vector tomada #1: compatível com Vector2d

A primeira versão de `Vector` deve ser tão compatível com nossa classe `Vector2d` anterior quanto possível.

No entanto, por design, o construtor de `Vector` não será compatível com o construtor de `Vector2d`. Poderíamos fazer `Vector(3, 4)` e `Vector(3, 4, 5)` funcionar aceitando argumentos quaisquer com `*args` em `__init__`, mas a melhor prática para o construtor de uma sequência é aceitar os dados como um argumento iterável no construtor, como o fazem todos os tipos embutidos de sequência. O exemplo 10.1 mostra algumas maneiras de instanciar objetos de nosso novo `Vector`.

Exemplo 10.1 – Testes de `Vector.__init__` e `Vector.__repr__`

```
>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

Exceto pela nova assinatura do construtor, garanti que todos os testes que fiz com `Vector2d` (por exemplo, `Vector2d(3, 4)`) passassem e produzissem o mesmo resultado de um `Vector([3, 4])` com dois componentes.



Quando um `Vector` tiver mais de seis componentes, a string gerada por `repr()` será abreviada com `...`, como vimos na última linha do exemplo 10.1. Isso é muito importante para qualquer tipo de coleção que possa conter uma grande quantidade de itens, pois `repr` é usado para depuração (e você não vai querer que um único objeto grande ocupe milhares de linhas de seu console ou log). Use o módulo `reprlib` para gerar representações de tamanho limitado, como mostra o exemplo 10.2. O módulo `reprlib` chama-se `repr` em Python 2. A ferramenta `2to3` reescreve as importações de `repr` automaticamente.

O exemplo 10.2 lista a implementação de nossa primeira versão de `Vector` (esse exemplo é baseado no código mostrado nos exemplos 9.2 e 9.3).

Exemplo 10.2 – `vector_v1.py`: derivado de `vector2d_v1.py`

```
from array import array
import reprlib
import math

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components) ❶
```



```

def __iter__(self):
    return iter(self._components) ❷

def __repr__(self):
    components = reprlib.repr(self._components) ❶
    components = components[components.find('['):-1] ❸
    return 'Vector({})'.format(components)

def __str__(self):
    return str(tuple(self))

def __bytes__(self):
    return (bytes([ord(self.typecode)]) +
            bytes(self._components)) ❺

def __eq__(self, other):
    return tuple(self) == tuple(other)

def __abs__(self):
    return math.sqrt(sum(x * x for x in self)) ❻

def __bool__(self):
    return bool(abs(self))

@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(memv) ❼

```

- ❶ O atributo de instância “protegido” `self._components` armazenará um array com os componentes de `Vector`.
- ❷ Para permitir iteração, devolvemos um iterador sobre `self._components`.¹
- ❸ Use `reprlib.repr()` para obter uma representação de tamanho limitado de `self._components` (por exemplo, `array('d', [0.0, 1.0, 2.0, 3.0, 4.0, ...])`).
- ❹ Remove o prefixo `array('d',` e o `)` final antes de inserir a string em uma chamada do construtor `Vector`.
- ❺ Cria um objeto `bytes` diretamente a partir de `self._components`.
- ❻ Não podemos mais usar `hypot`, então somamos os quadrados dos componentes e calculamos o `sqrt` desse valor.
- ❼ A única alteração necessária do `frombytes` anterior está na última linha: passamos a `memoryview` diretamente ao construtor sem desempacotar com `*`, como fazíamos antes.

¹ A função `iter()` será discutida no capítulo 14, juntamente com o método `__iter__`.

O modo como usei `reprlib.repr` merece alguns esclarecimentos. Essa função gera representações seguras de estruturas grandes ou recursivas limitando o tamanho da string de saída e marcando o ponto de corte com `'...'`. Eu quis que a `repr` de um `Vector` se parecesse com `Vector([3.0, 4.0, 5.0])`, e não com `Vector(array('d', [3.0, 4.0, 5.0]))`, pois o fato de haver um array em um `Vector` é um detalhe de implementação. Como essas chamadas ao construtor criam objetos `Vector` idênticos, prefiro a sintaxe mais simples, que usa um argumento `list`.

Ao escrever `__repr__`, eu poderia ter gerado a apresentação simplificada de `components` com esta expressão: `reprlib.repr(list(self._components))`. No entanto isso seria um desperdício, pois eu estaria copiando todos os itens de `self._components` para uma `list` somente para usar a `repr` de `list`. Em vez disso, decidi aplicar `reprlib.repr` diretamente ao array `self._components` e, em seguida, eliminar os caracteres fora dos `[]`. É isso que a segunda linha de `__repr__` faz no exemplo 10.2.



Por causa de seu papel na depuração, chamar `repr()` em um objeto jamais deve levantar uma exceção. Se algo der errado em sua implementação de `__repr__`, trate o problema e faça o melhor possível para gerar algum resultado útil que dê ao usuário uma chance de identificar o objeto-alvo.

Observe que os métodos `__str__`, `__eq__` e `__bool__` não sofreram alteração em relação a `Vector2d`, e apenas um caractere mudou em `frombytes` (um `*` foi removido da última linha). Essa é uma das vantagens de ter deixado o `Vector2d` original iterável.

A propósito, poderíamos ter feito de `Vector` uma subclasse de `Vector2d`, mas preferi não fazê-lo por dois motivos. Em primeiro lugar, por causa dos construtores incompatíveis, realmente não é aconselhável usar subclasses. Eu poderia ter contornado esse problema com algum tratamento esperto de parâmetros em `__init__`, mas o segundo motivo é mais importante: quero que `Vector` seja um exemplo de uma classe que implemente o protocolo de sequência sem herdar de alguma classe especial. É isso que faremos a seguir, após uma discussão sobre o termo *protocolo*.

Protocolos e duck typing

Desde o capítulo 1, vimos que não é preciso herdar de nenhuma classe especial para criar um tipo de sequência totalmente funcional em Python; basta implementar os métodos que atendam ao protocolo de sequência. Mas de que tipo de protocolo estamos falando?

No contexto de programação orientada a objetos, um protocolo é uma interface informal, definida somente na documentação, e não no código. Por exemplo, o protocolo de sequência em Python implica somente os métodos `__len__` e `__getitem__`. Qualquer

classe `Spam` que implemente esse métodos com a assinatura e a semântica-padrão pode ser usada em qualquer lugar em que se espera uma sequência. O fato de `Spam` ser uma subclasse dessa ou daquela classe é irrelevante: tudo que importa é que ela ofereça os métodos necessários. Vimos isso no exemplo 1.1, reproduzido aqui no exemplo 10.3.

Exemplo 10.3 – Código do exemplo 1.1 reproduzido aqui por conveniência

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

A classe `FrenchDeck` no exemplo 10.3 aproveita vários recursos de Python porque ela implementa o protocolo de sequência, mesmo que isso não esteja declarado em nenhum lugar no código. Qualquer programador Python experiente olhará para essa classe e entenderá que ela é uma sequência, apesar de ela ser uma subclasse de `object`. Dizemos que ela é uma sequência porque ela *se comporta* como tal, e é isso que importa.

Isso ficou conhecido como *duck typing* (tipagem pato), por causa do post de Alex Martelli, do qual a citação do início deste capítulo foi extraída.

Como os protocolos são informais e não são impostos, geralmente você pode implementar somente parte de um protocolo sem que haja problemas se conhecer o contexto específico em que a classe será usada. Por exemplo, para dar suporte à iteração, apenas `__getitem__` é necessário; não é preciso fornecer `__len__`.

Vamos agora implementar o protocolo de sequência em `Vector`, inicialmente sem um suporte apropriado para fatiamento, mas adicionando-o depois.

Vector tomada #2: uma sequência que permite fatiamento

Como vimos no exemplo com `FrenchDeck`, oferecer suporte ao protocolo de sequência é realmente fácil se você puder delegar a um atributo de sequência em seu objeto,

como o nosso array `self._components`. Estes métodos de uma linha, `__len__` e `__getitem__`, são um bom ponto de partida:

```
class Vector:
    # muitas linhas omitidas
    # ...

    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        return self._components[index]
```

Com esses acréscimos, todas as operações a seguir agora funcionam:

```
>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[-1]
(3.0, 5.0)
>>> v7 = Vector(range(7))
>>> v7[1:4]
array('d', [1.0, 2.0, 3.0])
```

Como você pode ver, temos suporte até mesmo para fatiamento – embora não esteja muito bom. Seria melhor se uma fatia de um `Vector` também fosse uma instância de `Vector`, e não um array. A velha classe `FrenchDeck` tem um problema parecido: ao fatiá-la, você obtém uma `list`. No caso de `Vector`, muitas funcionalidades são perdidas quando o fatiamento produz arrays simples.

Considere os tipos embutidos de sequência: todos eles, quando fatiados, produzem uma nova instância de seu próprio tipo, e não de outro tipo.

Para fazer `Vector` produzir fatias que sejam instâncias de `Vector`, não podemos simplesmente delegar o fatiamento a `array`. Devemos analisar os argumentos que recebemos em `__getitem__` e fazer o que é certo.

Vamos ver agora como Python transforma a sintaxe `my_seq[1:3]` em argumentos para `my_seq.__getitem__(...)`.

Como funciona o fatiamento

Uma demo vale por mil palavras, então vamos dar uma olhada no exemplo 104.

Exemplo 10.4 – Observando o comportamento de `__getitem__` e de fatias

```
>>> class MySeq:
...     def __getitem__(self, index):
...         return index # ❶
...
>>> s = MySeq()
>>> s[1] # ❷
1
>>> s[1:4] # ❸
slice(1, 4, None)
>>> s[1:4:2] # ❹
slice(1, 4, 2)
>>> s[1:4:2, 9] # ❺
(slice(1, 4, 2), 9)
>>> s[1:4:2, 7:9] # ❻
(slice(1, 4, 2), slice(7, 9, None))
```

❶ Para essa demonstração, `__getitem__` simplesmente devolve o que for passado para ele.

❷ Um único índice – nada de novo.

❸ A notação `1:4` transforma-se em `slice(1, 4, None)`.

❹ `slice(1, 4, 2)` quer dizer comece em 1, pare em 4, pule de 2 em 2.

❺ Surpresa: a presença de vírgulas em `[]` resulta que `__getitem__` recebe uma tupla.

❻ A tupla pode até mesmo armazenar vários objetos de fatia.

Vamos agora observar o próprio `slice` com mais detalhes no exemplo 10.5.

Exemplo 10.5 – Inspeccionando os atributos da classe `slice`

```
>>> slice # ❶
<class 'slice'>
>>> dir(slice) # ❷
['_class_', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'indices', 'start', 'step', 'stop']
```

❶ `slice` é um tipo embutido (nós o vimos pela primeira vez na seção “Objetos `slice`” na página 60).

- ❷ Ao inspecionar um slice, encontramos os atributos de dados `start`, `stop` e `step` e um método `indices`.

No exemplo 10.5, chamar `dir(slice)` revela um atributo `indices`, que, por acaso, é um método bem interessante, mas pouco conhecido. Eis o que `help(slice.indices)` revela:

```
S.indices(len) -> (start, stop, stride)
```

Considerando uma sequência de tamanho `len`, calcula os índices `start` e `stop` e o tamanho de `stride` da fatia estendida descrita por `S`. Índices fora dos limites serão reduzidos de modo consistente com o tratamento de fatias normais.

Em outras palavras, `indices` expõe a lógica complicada implementada em sequências embutidas para tratar índices ausentes ou negativos e fatias maiores que a sequência-alvo de forma elegante. Esse método produz tuplas “normalizadas” de inteiros não negativos – `start`, `stop` e `stride` – ajustados para que estejam dentro dos limites de uma sequência do tamanho especificado.

Eis alguns exemplos, considerando uma sequência de `len == 5`, por exemplo, `'ABCDE'`:

```
>>> slice(None, 10, 2).indices(5) # ❶
(0, 5, 2)
>>> slice(-3, None, None).indices(5) # ❷
(2, 5, 1)
```

❶ `'ABCDE'[:10:2]` é igual a `'ABCDE'[0:5:2]`

❷ `'ABCDE'[-3:]` é igual a `'ABCDE'[2:5:1]`



Quando escrevi este texto, o método `slice.indices` aparentemente não estava documentado online em Python Library Reference (Guia de referência à biblioteca Python). O *Python Python/C API Reference Manual* (Manual de referência da API Python/C de Python) documenta uma função semelhante em C, `PySlice_GetIndicesEx` (https://docs.python.org/3/c-api/slice.html#c.PySlice_GetIndicesEx). Descobri `slice.indices` enquanto explorava objetos `slice` no console de Python usando `dir()` e `help()`. É outra evidência da importância do console interativo como ferramenta para descobertas.

No código de nosso `Vector`, não precisaremos do método `slice.indices()` porque, quando recebemos um argumento que é uma fatia, delegamos seu tratamento a `array_components`. Mas se você não puder contar com os serviços de uma sequência subjacente, esse método permitirá economizar bastante tempo.

Agora que você sabe como tratar fatias, vamos dar uma olhada na implementação melhorada de `Vector.__getitem__`.

Um `__getitem__` que considera fatias

O exemplo 10.6 lista os dois métodos necessários para fazer `Vector` comportar-se como uma sequência: `__len__` e `__getitem__` (o último está implementado agora para tratar fatiamento de forma correta).

Exemplo 10.6 – Parte de `vector_v2.py`: métodos `__len__` e `__getitem__` adicionados à classe `Vector` de `vector_v1.py` (veja o exemplo 10.2)

```
def __len__(self):
    return len(self._components)

def __getitem__(self, index):
    cls = type(self) ❶
    if isinstance(index, slice): ❷
        return cls(self._components[index]) ❸
    elif isinstance(index, numbers.Integral): ❹
        return self._components[index] ❺
    else:
        msg = '{cls.__name__} indices must be integers'
        raise TypeError(msg.format(cls=cls)) ❻
```

- ❶ Obtém a classe da instância (isto é, `Vector`) para usar depois.
- ❷ Se o argumento `index` for um `slice`...
- ❸ ... chama a classe para criar outra instância de `Vector` a partir de uma fatia do array `_components`.
- ❹ Se `index` for um `int` ou outro tipo de inteiro...
- ❺ ... basta retornar o item específico de `_components`.
- ❻ Caso contrário, levanta uma exceção.



O uso excessivo de `isinstance` pode ser sinal de um design orientado a objetos ruim, mas tratar fatias em `__getitem__` é um caso de uso em que isso se justifica. No exemplo 10.6, observe o teste em relação a `numbers.Integral` – uma `Abstract Base Class` (Classe Base Abstrata). Usar ABCs em testes com `isinstance` deixa uma API mais flexível e preparada para o futuro. O capítulo 11 explica por quê. Infelizmente, não há uma ABC para `slice` na biblioteca-padrão de Python 3.4.

Para descobrir qual exceção levantar na cláusula `else` de `__getitem__`, usei o console interativo para verificar o resultado de `'ABC'[1, 2]`. Então vi que Python levanta um `TypeError` e copiei também o texto da mensagem de erro: “indices must be integers” (índices devem ser inteiros). Para criar objetos pythônicos, espelhe-se nos próprios objetos de Python.

Depois que o código do exemplo 10.6 for adicionado à classe `Vector`, teremos um comportamento apropriado para fatiamento, como mostra o exemplo 10.7.

Exemplo 10.7 – Testes de `Vector.getitem` melhorado do exemplo 10.6.

```
>>> v7 = Vector(range(7))
>>> v7[-1] ❶
6.0
>>> v7[1:4] ❷
Vector([1.0, 2.0, 3.0])
>>> v7[-1:] ❸
Vector([6.0])
>>> v7[1,2] ❹
Traceback (most recent call last):
...
```

```
TypeError: Vector indices must be integers
```

- ❶ Um índice inteiro recupera apenas o valor de um componente como um float.
- ❷ Um índice de fatia cria um novo `Vector`.
- ❸ Uma fatia de `len == 1` também cria um `Vector`.
- ❹ `Vector` não aceita indexação multidimensional, portanto uma tupla de índices ou fatias levanta um erro.

Vector tomada #3: acesso dinâmico a atributos

Na evolução de `Vector2d` para `Vector`, perdemos a capacidade de acessar os componentes do vetor pelo nome (por exemplo, `v.x`, `v.y`). Estamos agora lidando com vetores que podem ter um grande número de componentes. Apesar disso, poderá ser conveniente acessar os primeiros componentes com letras simples como `x`, `y`, `z` em vez de usar `v[0]`, `v[1]` e `v[2]`.

Eis a sintaxe alternativa que queremos oferecer para ler os quatro primeiros componentes de um vetor:

```
>>> v = Vector(range(10))
>>> v.x
0.0
>>> v.y, v.z, v.t
(1.0, 2.0, 3.0)
```

Em `Vector2d`, oferecemos acesso somente de leitura para `x` e `y` usando o decorador `@property` (Exemplo 9.7). Poderíamos criar quatro propriedades em `Vector`, mas isso seria tedioso. O método especial `__getattr__` oferece uma opção melhor.

O método `__getattr__` é chamado pelo interpretador quando a busca pelo atributo falha. Falando de modo simplificado, dada a expressão `my_obj.x`, Python verifica se a instância `my_obj` tem um atributo chamado `x`; se não tiver, a busca passa para a classe (`my_obj.__class__`) e, então, para o grafo de herança.² Se o atributo `x` não for encontrado, o método `__getattr__` definido na classe de `my_obj` será chamado com `self` e o nome do atributo, como uma string (por exemplo, `'x'`).

O exemplo 10.8 mostra nosso método `__getattr__`. Essencialmente, ele verifica se o atributo procurado é uma das letras `xyzt`; se for, esse método devolve o componente correspondente do vetor.

Exemplo 10.8 – Parte de `vector_v3.py`: método `__getattr__` adicionado à classe `Vector` de `vector_v2.py`

```
shortcut_names = 'xyzt'

def __getattr__(self, name):
    cls = type(self) ❶
    if len(name) == 1: ❷
        pos = cls.shortcut_names.find(name) ❸
        if 0 <= pos < len(self._components): ❹
            return self._components[pos]
    msg = '{.__name__!r} object has no attribute {!r}' ❺
    raise AttributeError(msg.format(cls, name))
```

- ❶ Obtém a classe `Vector` para usar depois.
- ❷ Se o nome tiver um caractere, poderá estar em `shortcut_names`.
- ❸ Encontra a posição do nome de uma só letra; `str.find` também localizará `'yz'`, mas não queremos isso; esse é o motivo do teste anterior.
- ❹ Se a posição estiver no intervalo válido, devolve o elemento do array.
- ❺ Se algum dos testes falhar, levanta `AttributeError` com um texto de mensagem-padrão.

Não é difícil implementar `__getattr__`, mas nesse caso não é suficiente. Considere a interação bizarra do exemplo 10.9.

Exemplo 10.9 – Comportamento inapropriado: atribuição a `v.x` não gera erro, mas introduz uma inconsistência

```
>>> v = Vector(range(5))
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])
>>> v.x # ❶
```

² A busca de atributos é mais complicada que isso; veremos os detalhes intrincados na Parte VI. Por enquanto, essa explicação simplificada deve bastar.

```
0.0
>>> v.x = 10 # ❷
>>> v.x # ❸
10
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0]) # ❹
```

- ❶ Acessa o elemento `v[0]` como `v.x`.
- ❷ Atribui novo valor a `v.x`. Isso deveria levantar uma exceção.
- ❸ Ler `v.x` mostra o novo valor, 10.
- ❹ Mas os componentes do vetor não mudaram.

Você consegue explicar o que está acontecendo? Em particular, por que, na segunda vez, `v.x` devolve 10 se esse valor não está no array de componentes do vetor? Se você não sabe de imediato, estude a explicação dada a `__getattr__` imediatamente antes do exemplo 10.8. É um pouco sutil, mas é uma base muito importante para entender muito do que virá depois no livro.

A inconsistência no exemplo 10.9 foi introduzida por causa do modo como `__getattr__` funciona: Python chama esse método somente como alternativa, quando o objeto não tiver o atributo especificado. Entretanto, após fazer a atribuição `v.x = 10`, o objeto `v` agora tem um atributo `x`, portanto `__getattr__` não será mais chamado para recuperar `v.x`: o interpretador simplesmente devolverá o valor 10 associado a `v.x`. Por outro lado, nossa implementação de `__getattr__` não presta atenção em atributos de instância que não sejam `self._components`, de onde ele recupera os valores dos “atributos virtuais” listados em `shortcut_names`.

Precisamos personalizar a lógica para definição de atributos em nossa classe `Vector` para evitar essa inconsistência.

Lembre-se de que, nos últimos exemplos de `Vector2d` do capítulo 9, a tentativa de atribuir valores aos atributos de instância `.x` ou `.y` levantaram `AttributeError`. Em `Vector`, queremos a mesma exceção nas tentativas de dar valor a qualquer atributo com uma única letra minúscula, apenas para evitar confusão. Para isso, implementaremos `__setattr__` conforme mostrado no exemplo 10.10.

Exemplo 10.10 – Parte de `vector_v3.py`: método `__setattr__` na classe `Vector`

```
def __setattr__(self, name, value):
    cls = type(self)
    if len(name) == 1: ❶
        if name in cls.shortcut_names: ❷
            error = 'readonly attribute {attr_name!r}'
```

```
elif name.islower(): ❸
    error = "can't set attributes 'a' to 'z' in {cls_name!r}"
else:
    error = '' ❹
if error: ❺
    msg = error.format(cls_name=cls.__name__, attr_name=name)
    raise AttributeError(msg)
super().__setattr__(name, value) ❻
```

- ❶ Tratamento especial para nomes de atributo com um único caractere.
- ❷ Se `name` estiver em `xyzt`, define uma mensagem de erro específica.
- ❸ Se `name` usar letras minúsculas, define a mensagem de erro relacionada a todos os nomes com uma única letra.
- ❹ Caso contrário, define uma mensagem de erro em branco.
- ❺ Se houver uma mensagem de erro que não esteja em branco, levanta `AttributeError`.
- ❻ Caso default: chama `__setattr__` da superclasse para ter um comportamento-padrão.



A função `super()` oferece uma maneira de acessar métodos de superclasses dinamicamente: uma necessidade em uma linguagem dinâmica que suporte herança múltipla, como Python. Ela é usada para delegar algumas tarefas de um método de uma subclasse para um método apropriado em uma superclasse, como vimos no exemplo 10.10. Há mais informações sobre `super` na seção “Herança múltipla e ordem de resolução de métodos” na página 399.

Ao escolher a mensagem de erro a ser exibida com `AttributeError`, verifiquei primeiro o comportamento do tipo embutido `complex`, pois ele é imutável e tem um par de atributos de dados `real` e `imag`. Tentar alterar qualquer um deles em uma instância de `complex` levanta `AttributeError` com a mensagem “can't set attribute”. Por outro lado, a tentativa de atribuir valor a um atributo somente de leitura protegido por uma propriedade, como fizemos na seção “Um `Vector2d` hashable” na página 298, produz a mensagem “readonly attribute”. Busquei inspiração nos dois textos para definir a string `error` em `__setitem__`, mas fui mais explícito sobre os atributos proibidos.

Observe que não estamos proibindo a atribuição de valores em todos os atributos, mas apenas naqueles com uma única letra minúscula, para evitar confusão com os atributos `x`, `y`, `z` e `t` somente de leitura.



Sabendo que declarar `__slots__` no nível de classe impede a definição de novos atributos de instância, usar esse recurso em vez de implementar `__setattr__` como fizemos é tentador. Contudo, por causa de todas as ressalvas discutidas na seção “Os problemas com `__slots__`” na página 309, o uso de `__slots__` somente para impedir a criação de atributos de instância não é recomendável. `__slots__` deve ser usado apenas para economizar memória, e somente se isso for realmente um problema.

Mesmo sem suporte à escrita nos componentes de `Vector`, eis uma lição importante desse exemplo: com muita frequência, ao implementar `__getattr__`, você deverá escrever `__setattr__` também para evitar um comportamento inconsistente em seus objetos.

Se quiséssemos permitir a alteração de componentes do vetor, poderíamos implementar `__setitem__` para permitir `v[0] = 1.1` e/ou `__setattr__` para fazer `v.x = 1.1` funcionar. Mas `Vector` permanecerá imutável porque queremos torná-lo hashable na próxima seção.

Vector tomada #4: hashing e um `==` mais rápido

Mais uma vez, precisamos implementar um método `__hash__`. Juntamente com o `__eq__` presente, isso deixará as instâncias de `Vector` hashable.

`__hash__` no exemplo 9.8 simplesmente calculava `hash(self.x) ^ hash(self.y)`. Agora queremos aplicar o operador `^` (xor) aos hashes de todos os componentes, sucessivamente, assim: `v[0] ^ v[1] ^ v[2]...`. É para isso que serve a função `functools.reduce`. Anteriormente, eu disse que `reduce` não é tão popular quanto costumava ser,³ mas calcular o hash de todos os componentes do vetor é um trabalho perfeito para ela. A figura 10.1 representa a ideia geral da função `reduce`.

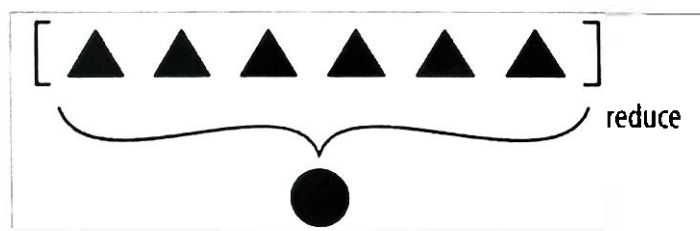


Figura 10.1 – Funções de redução – `reduce`, `sum`, `any`, `all` – produzem um único resultado agregado a partir de uma sequência ou de qualquer objeto iterável finito.

Até agora, vimos que `functools.reduce()` pode ser substituído por `sum()`, mas vamos explicar devidamente o seu funcionamento. A ideia principal é reduzir uma série de valores a um único valor. O primeiro argumento de `reduce()` é uma função de dois argumentos, e o segundo argumento é um iterável. Vamos supor que temos uma

³ `sum`, `any` e `all` englobam os usos mais comuns de `reduce`. Veja a discussão na seção “Substitutos modernos para `map`, `filter` e `reduce`” na página 178.

função `fn` de dois argumentos e uma lista `lst`. Ao chamar `reduce(fn, lst)`, `fn` será aplicado ao primeiro par de elementos – `fn(lst[0], lst[1])` – produzindo o primeiro resultado, `r1`. Em seguida, `fn` é aplicado a `r1` e ao próximo elemento – `fn(r1, lst[2])` –, produzindo um segundo resultado, `r2`. Agora `fn(r2, lst[3])` será chamado para produzir `r3` ..., e assim sucessivamente, até o último elemento, quando um único resultado, `rN`, será devolvido.

Você pode usar `reduce` da seguinte maneira para calcular 5! (o fatorial de 5):

```
>>> 2 * 3 * 4 * 5 # o resultado que queremos: 5! == 120
120
>>> import functools
>>> functools.reduce(lambda a,b: a*b, range(1, 6))
120
```

De volta ao nosso problema de hashing, o exemplo 10.11 mostra a ideia de calcular o xor agregado fazendo isso de três maneiras: com um laço `for` e duas chamadas a `reduce`.

Exemplo 10.11 – Três maneiras de calcular o xor acumulado de inteiros de 0 a 5

```
>>> n = 0
>>> for i in range(1, 6): # ❶
...     n ^= i
...
>>> n
1
>>> import functools
>>> functools.reduce(lambda a, b: a^b, range(6)) # ❷
1
>>> import operator
>>> functools.reduce(operator.xor, range(6)) # ❸
1
```

❶ xor agregado com um laço `for` e uma variável acumuladora.

❷ `functools.reduce` usando uma função anônima.

❸ `functools.reduce` substituindo o `lambda` por `operator.xor`.

Das alternativas no exemplo 10.11, a última é minha favorita, e o laço `for` vem depois. Qual delas você prefere?

Como vimos na seção “Módulo `operator`” na página 194, `operator` oferece a funcionalidade de todos os operadores infixos de Python em forma de função, reduzindo a necessidade de usar `lambda`.

Para implementar `Vector.__hash__` em meu estilo predileto, precisamos importar os módulos `functools` e `operator`. O exemplo 10.12 mostra as alterações relevantes.

Exemplo 10.12 – Parte de `vector_v4.py`: duas importações e o método `__hash__` adicionados à classe `Vector` de `vector_v3.py`

```
from array import array
import reprlib
import math
import functools # ❶
import operator # ❷

class Vector:
    typecode = 'd'

    # muitas linhas omitidas na listagem do livro...

    def __eq__(self, other): # ❸
        return tuple(self) == tuple(other)

    def __hash__(self):
        hashes = (hash(x) for x in self._components) # ❹
        return functools.reduce(operator.xor, hashes, 0) # ❺

    # mais linhas omitidas...
```

- ❶ Importa `functools` para usar `reduce`.
- ❷ Importa `operator` para usar `xor`.
- ❸ Nenhuma alteração em `__eq__`; listei-o aqui porque manter `__eq__` e `__hash__` próximos no código-fonte é uma boa prática, pois eles devem trabalhar juntos.
- ❹ Cria uma expressão geradora para calcular o hash de cada componente em modo lazy (por demanda).
- ❺ Alimenta `reduce` com hashes usando a função `xor` para calcular o valor de hash agregado; o terceiro argumento, `0`, é o inicializador (veja o aviso a seguir).



Ao usar `reduce`, é uma boa prática fornecer o terceiro argumento, `reduce(function, iterable, initializer)`, para evitar esta exceção: `TypeError: reduce() of empty sequence with no initial value` [(`TypeError: reduce() de sequência vazia sem valor inicial`); é uma mensagem excelente: explica o problema e como corrigi-lo]. `initializer` é o valor devolvido se a sequência for vazia e é usado como primeiro argumento no laço de redução, portanto deve ser o valor de identidade da operação. Por exemplo, para `+`, `|` e `^`, `initializer` deve ser `0`, mas para `*` e `&`, deve ser `1`.

Conforme implementado, o método `__hash__` no exemplo 10.8 é um exemplo perfeito de uma computação map-reduce (Figura 10.2).

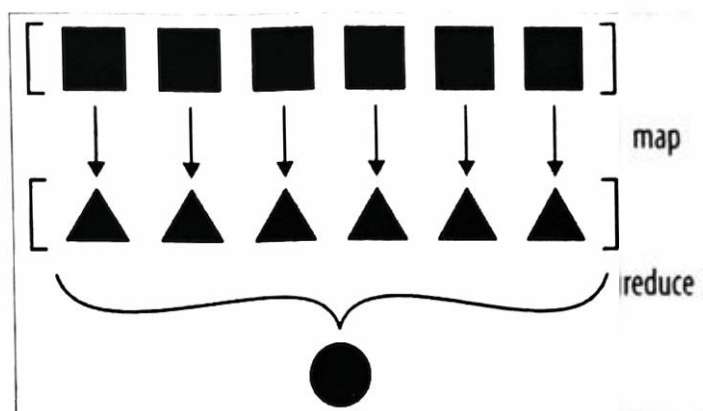


Figura 10.2 – Map-reduce: aplica a função a cada item para gerar uma nova série (map) e, em seguida, calcula um valor agregado (reduce).

O passo do mapeamento gera um hash para cada componente, e o passo de redução agrega todos os hashes com o operador `xor`. Usar `map` em vez de uma *genexp* deixa o passo de mapeamento mais visível:

```
def __hash__(self):
    hashes = map(hash, self._components)
    return functools.reduce(operator.xor, hashes)
```



A solução com `map` é menos eficiente em Python 2, em que a função `map` cria uma nova list com os resultados. Contudo, em Python 3, `map` é lazy: ele cria um gerador que produz os resultados por demanda, economizando memória – como a expressão geradora que usamos no método `__hash__` do exemplo 10.8.

Enquanto estamos discutindo funções de redução, podemos substituir nossa implementação simples de `__eq__` por outra que será menos custosa em termos de processamento e de memória, pelo menos para vetores grandes. Como apresentada no exemplo 9.2, temos a seguinte implementação bem concisa de `__eq__`:

```
def __eq__(self, other):
    return tuple(self) == tuple(other)
```

Isso funciona para `Vector2d` e `Vector` – ela até mesmo considera `Vector([1, 2])` igual a `(1, 2)`, o que pode ser um problema, mas vamos ignorar isso por enquanto.⁴ No entanto, para instâncias de `Vector` que podem ter milhares de componentes, esse método é muito ineficiente. Ele cria duas tuplas copiando todo o conteúdo dos operandos somente para usar o `__eq__` do tipo `tuple`. Para `Vector2d` (com apenas dois componentes), é um bom atalho, porém não para vetores multidimensionais grandes.

⁴ Vamos considerar seriamente a questão de `Vector([1, 2]) == (1, 2)` na seção “Básico da sobrecarga de operadores” na página 420.

Uma maneira melhor de comparar um `Vector` com outro `Vector` ou com um iterável seria o código do exemplo 10.13.

Exemplo 10.13 – `Vector.eq` usando `zip` em um laço `for` para comparações mais eficientes

```
def __eq__(self, other):
    if len(self) != len(other): # ❶
        return False
    for a, b in zip(self, other): # ❷
        if a != b: # ❸
            return False
    return True # ❹
```

- ❶ Se o `len` dos objetos for diferente, é sinal de que eles não são iguais.
- ❷ `zip` produz um gerador de tuplas composto a partir dos itens de cada argumento iterável. Veja a caixa de texto “O fantástico `zip`” a seguir se você ainda não conhece `zip`. A comparação anterior com `len` é necessária porque `zip` para de produzir valores sem avisar assim que uma das entradas se esgota.
- ❸ Assim que dois componentes forem diferentes, sai devolvendo `False`.
- ❹ Caso contrário, os objetos são iguais.

O exemplo 10.13 é eficiente, mas a função `all` pode produzir o mesmo cálculo de agregação do laço `for` em uma linha: se todas as comparações entre os componentes correspondentes dos operandos for `True`, o resultado será `True`. Assim que uma comparação for `False`, `all` devolverá `False`. O exemplo 10.14 mostra `__eq__` usando `all`.

Exemplo 10.14 – `Vector.eq` usando `zip` e `all`: mesma lógica do exemplo 10.13

```
def __eq__(self, other):
    return len(self) == len(other) and all(a == b for a, b in zip(self, other))
```

Observe que, em primeiro lugar, verificamos se os operandos têm tamanhos iguais, pois `zip` para no menor operando.

O exemplo 10.14 mostra a implementação que escolhemos para `__eq__` em `vector_v4.py`.

Encerraremos este capítulo trazendo de volta o método `__format__` de `Vector2d` para `Vector`.

O fantástico `zip`

Ter um laço `for` que faz a iteração pelos itens sem lidar com variáveis de índice é ótimo e evita muitos bugs, mas exige algumas funções utilitárias especiais. Uma delas é a função embutida `zip`, que facilita a iteração em paralelo em dois ou mais

iteráveis devolvendo tuplas que você pode desempacotar em variáveis, uma para cada item das entradas paralelas. Veja o exemplo 10.15.



A função `zip` deve seu nome ao zíper porque o mecanismo físico funciona pelo travamento de pares de dentes dos dois lados do zíper, e é uma boa analogia visual para o que `zip(esquerda, direita)` faz. Não há nenhuma relação com arquivos compactados.

Exemplo 10.15 – A função embutida `zip` em funcionamento

```
>>> zip(range(3), 'ABC') # ❶
<zip object at 0x10063ae48>
>>> list(zip(range(3), 'ABC')) # ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(zip(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3])) # ❸
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2)]
>>> from itertools import zip_longest # ❹
>>> list(zip_longest(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3], fillvalue=-1))
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2), (-1, -1, 3.3)]
```

- ❶ `zip` devolve um gerador que produz tuplas por demanda.
- ❷ Criamos uma `list` com ele somente para exibição; normalmente, iteramos pelo gerador.
- ❸ `zip` tem uma característica surpreendente: ela para sem avisar quando um dos iteráveis se esgota.⁵
- ❹ A função `itertools.zip_longest` comporta-se de modo diferente: ela usa um `fillvalue` opcional (`None` por default) para completar valores ausentes a fim de poder gerar tuplas até que o último iterável se esgote.

A função embutida `enumerate` é outra função geradora, usada com frequência em laços `for` para evitar o tratamento manual de variáveis de índice. Se você não tem familiaridade com `enumerate`, dê uma olhada sem falta na documentação em “Built-in functions” (Funções embutidas, <http://bit.ly/1QOtsk8>). As funções embutidas `zip` e `enumerate`, juntamente com várias outras funções geradoras da biblioteca-padrão, serão discutidas na seção “Funções geradoras da biblioteca-padrão” na página 474.

⁵ Isso é surpreendente (pelo menos, para mim). Acho que `zip` deveria levantar `ValueError` se as sequências não tiverem o mesmo tamanho, que é o que acontece quando desempacotamos um iterável em uma tupla de variáveis de tamanhos diferentes.

Vector tomada #5: formatação

O método `__format__` de `Vector` lembrará o de `Vector2d`, mas, em vez de oferecer uma apresentação personalizada em coordenadas polares, `Vector` usará coordenadas esféricas – também conhecidas como coordenadas “hiperesféricas”, pois agora aceitamos n dimensões, e as esferas são “hiperesferas” de 4D em diante.⁶ Para estar de acordo com isso, mudaremos o sufixo personalizado de formatação de 'p' para 'h'.



Como vimos na seção “Apresentações formatadas” na página 295, ao estender a Minilinguagem de Especificação de Formatação (Format Specification Mini-Language, <https://docs.python.org/3/library/string.html#formatspec>), é melhor evitar a reutilização de códigos de formatação aceitos por tipos embutidos. Em particular, nossa minilinguagem estendida também usa os códigos de formatação de ponto flutuante 'eEfFgGn%' em seu sentido original, portanto, definitivamente, devemos evitá-los. Inteiros usam 'bcdxXn' e strings usam 's'. Escolhi 'p' para coordenadas polares de `Vector2d`. O código 'h' para coordenadas hiperesféricas é uma boa opção.

Por exemplo, dado um objeto `Vector` no espaço 4D (`len(v) == 4`), o código 'h' produzirá uma apresentação como `<r, ϕ_1 , ϕ_2 , ϕ_3 >`, em que r é a magnitude (`abs(v)`) e os números restantes são as coordenadas angulares ϕ_1 , ϕ_2 , ϕ_3 .

Eis alguns exemplos de formatação de coordenadas esféricas em 4D, extraídas dos doctests de `vector_v5.py` (veja o exemplo 10.16):

```
>>> format(Vector([-1, -1, -1, -1]), 'h')
'<2.0, 2.0943951023931957, 2.186276035465284, 3.9269908169872414>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
```

Antes de podermos implementar as pequenas alterações necessárias em `__format__`, devemos escrever um par de métodos auxiliares: `angle(n)` para calcular uma das coordenadas angulares (por exemplo, ϕ_1), e `angles()` para devolver um iterável de todas as coordenadas angulares. Não descreverei a matemática aqui: se estiver curioso, a entrada “ n -sphere” (n -esfera, <http://en.wikipedia.org/wiki/N-sphere>) na Wikipedia tem as fórmulas que usei para calcular as coordenadas esféricas a partir das coordenadas cartesianas do array de componentes de `Vector`.

⁶ O site Wolfram Mathworld tem um artigo sobre hiperesferas (Hypersphere, <http://mathworld.wolfram.com/Hypersphere.html>); na Wikipedia, “hypersphere” é redirecionado para a entrada “ n -sphere” (<http://en.wikipedia.org/wiki/N-sphere>).

O exemplo 10.16 apresenta uma listagem completa de *vector_v5.py*, consolidando tudo que implementamos desde a seção “Vector tomada #1: compatível com Vector2d” na página 319 e introduzindo a formatação personalizada.

Exemplo 10.16 – *vector_v5.py*: doctests e todo o código da classe Vector final; os comentários numerados destacam os acréscimos necessários para suporte a `__format__`

```
"""
```

Uma classe ``Vector`` multidimensional, tomada 5

Um ``Vector`` é criado a partir de um iterável de números::

```
>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

Testes com duas dimensões (mesmo resultado de `vector2d_v1.py`)::

```
>>> v1 = Vector([3, 4])
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector([3.0, 4.0])
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector([0, 0]))
(True, False)
```

Teste do método de classe `Vector.frombytes()`:

```
>>> v1_clone = Vector.frombytes(bytes(v1))
>>> v1_clone
Vector([3.0, 4.0])
```

```
>>> v1 == v1_clone
True
```

Testes com três dimensões::

```
>>> v1 = Vector([3, 4, 5])
>>> x, y, z = v1
>>> x, y, z
(3.0, 4.0, 5.0)
>>> v1
Vector([3.0, 4.0, 5.0])
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0, 5.0)
>>> abs(v1) # doctest:+ELLIPSIS
7.071067811...
>>> bool(v1), bool(Vector([0, 0, 0]))
(True, False)
```

Testes com várias dimensões::

```
>>> v7 = Vector(range(7))
>>> v7
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
>>> abs(v7) # doctest:+ELLIPSIS
9.53939201...
```

Teste dos métodos ``.__bytes__`` e ``.frombytes()``::

```
>>> v1 = Vector([3, 4, 5])
>>> v1_clone = Vector.frombytes(bytes(v1))
>>> v1_clone
Vector([3.0, 4.0, 5.0])
>>> v1 == v1_clone
True
```

Testes de comportamento de sequência::

```
>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[len(v1)-1], v1[-1]
(3.0, 5.0, 5.0)
```

Testes de fatiamento::

```
>>> v7 = Vector(range(7))
>>> v7[-1]
6.0
>>> v7[1:4]
Vector([1.0, 2.0, 3.0])
>>> v7[-1:]
Vector([6.0])
>>> v7[1,2]
Traceback (most recent call last):
...
TypeError: Vector indices must be integers
```

Testes de acesso dinâmico a atributos::

```
>>> v7 = Vector(range(10))
>>> v7.x
0.0
>>> v7.y, v7.z, v7.t
(1.0, 2.0, 3.0)
```

Falhas em busca dinâmica de atributos::

```
>>> v7.k
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 'k'
>>> v3 = Vector(range(3))
>>> v3.t
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 't'
>>> v3.spam
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 'spam'
```

Testes de hashing:

```
>>> v1 = Vector([3, 4])
>>> v2 = Vector([3.1, 4.2])
>>> v3 = Vector([3, 4, 5])
>>> v6 = Vector(range(6))
>>> hash(v1), hash(v3), hash(v6)
```



```
(7, 2, 1)
```

A maioria dos valores de hash de não inteiros varia entre uma versão de CPython de 32 ou de 64 bits::

```
>>> import sys
>>> hash(v2) == (384307168202284039 if sys.maxsize > 2**32 else 357915986)
True
```

Testes de ``format()`` com coordenadas cartesianas em 2D:

```
>>> v1 = Vector([3, 4])
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

Testes de ``format()`` com coordenadas cartesianas em 3D e 7D:

```
>>> v3 = Vector([3, 4, 5])
>>> format(v3)
'(3.0, 4.0, 5.0)'
>>> format(Vector(range(7)))
'(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)'
```

Testes de ``format()`` com coordenadas esféricas em 2D, 3D e 4D::

```
>>> format(Vector([1, 1]), 'h') # doctest:+ELLIPSIS
'<1.414213..., 0.785398...>'
>>> format(Vector([1, 1]), '.3eh')
'<1.414e+00, 7.854e-01>'
>>> format(Vector([1, 1]), '0.5fh')
'<1.41421, 0.78540>'
>>> format(Vector([1, 1, 1]), 'h') # doctest:+ELLIPSIS
'<1.73205..., 0.95531..., 0.78539...>'
>>> format(Vector([2, 2, 2]), '.3eh')
'<3.464e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 0, 0]), '0.5fh')
'<0.00000, 0.00000, 0.00000>'
>>> format(Vector([-1, -1, -1, -1]), 'h') # doctest:+ELLIPSIS
'<2.0, 2.09439..., 2.18627..., 3.92699...>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
```

```

>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
"""

from array import array
import reprlib
import math
import numbers
import functools
import operator
import itertools ❶

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        return iter(self._components)

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('{'):-1]
        return 'Vector({})'.format(components)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))

    def __hash__(self):
        hashes = (hash(x) for x in self)
        return functools.reduce(operator.xor, hashes, 0)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __bool__(self):
        return bool(abs(self))

    def __len__(self):
        return len(self._components)

```

```

def __getitem__(self, index):
    cls = type(self)
    if isinstance(index, slice):
        return cls(self._components[index])
    elif isinstance(index, numbers.Integral):
        return self._components[index]
    else:
        msg = '{.__name__} indices must be integers'
        raise TypeError(msg.format(cls))

shortcut_names = 'xyzt'

def __getattr__(self, name):
    cls = type(self)
    if len(name) == 1:
        pos = cls.shortcut_names.find(name)
        if 0 <= pos < len(self._components):
            return self._components[pos]
        msg = '{.__name__!r} object has no attribute {!r}'
        raise AttributeError(msg.format(cls, name))

def angle(self, n): ❷
    r = math.sqrt(sum(x * x for x in self[n:]))
    a = math.atan2(r, self[n-1])
    if (n == len(self) - 1) and (self[-1] < 0):
        return math.pi * 2 - a
    else:
        return a

def angles(self): ❸
    return (self.angle(n) for n in range(1, len(self)))

def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('h'): # coordenadas hiperesféricas
        fmt_spec = fmt_spec[:-1]
        coords = itertools.chain([abs(self)],
                                self.angles()) ❹
        outer_fmt = '<{}>' ❺
    else:
        coords = self
        outer_fmt = '({})' ❻
    components = (format(c, fmt_spec) for c in coords) ❼
    return outer_fmt.format(', '.join(components)) ❶

```

```
@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(memv)
```

- ❶ Importa `itertools` para usar a função `chain` em `__format__`.
- ❷ Calcula uma das coordenadas angulares usando fórmulas adaptadas do artigo sobre *n-esfera* (*n-sphere*, <http://en.wikipedia.org/wiki/N-sphere>).
- ❸ Cria expressão geradora para calcular todas as coordenadas angulares por demanda.
- ❹ Usa `itertools.chain` para produzir a *genexp* e iterar naturalmente pela magnitude e as coordenadas angulares.
- ❺ Configura a apresentação de coordenadas esféricas com sinais de “menor que” e “maior que”.
- ❻ Configura a apresentação de coordenadas cartesianas com parênteses.
- ❼ Cria uma expressão geradora para formatar cada item de coordenada por demanda.
- ❽ Insere componentes formatados, separados por vírgulas, entre colchetes ou parênteses.



Estamos fazendo uso intensivo de expressões geradoras em `__format__`, `angle` e `angles`, porém nosso foco aqui é codar `__format__` para trazer o mesmo nível de implementação de `Vector2d` a `Vector`. Quando abordarmos os geradores no capítulo 14, usaremos parte do código de `Vector` como exemplos e, nessa ocasião, os truques com geradores serão explicados com detalhes.

Com isso, concluímos nossa missão para este capítulo. A classe `Vector` será expandida com operadores infixos no capítulo 13, mas nosso objetivo aqui foi explorar técnicas para criação de métodos especiais úteis em uma grande variedade de classes de coleção.

Resumo do capítulo

O exemplo com `Vector` neste capítulo foi criado para que fosse compatível com `Vector2d`, exceto pelo uso de uma assinatura diferente de construtor, que aceita um único argumento iterável, exatamente como fazem os tipos embutidos de sequência. O fato de `Vector` se comportar como uma sequência somente por implementar `__getitem__` e `__len__` motivou uma discussão sobre protocolos, que são as interfaces informais usadas em linguagens com duck typing.

Em seguida, vimos como a sintaxe `my_seq[a:b:c]` funciona internamente criando um objeto `slice(a, b, c)` e passando-o para `__getitem__`. De posse desse conhecimento, fizemos `Vector` responder corretamente ao fatiamento, devolvendo novas instâncias de `Vector`, exatamente como se espera de uma sequência `pythônica`.

O próximo passo foi proporcionar acesso somente de leitura aos primeiros componentes de `Vector` usando uma notação como `my_vec.x`. Fizemos isso implementando `__getattr__`. Fazer isso abriu a possibilidade tentadora de o usuário fazer atribuições a esses componentes especiais escrevendo `my_vec.x = 7`, trazendo à tona um bug em potencial. Corrigimos esse bug implementando `__setattr__` também para proibir a atribuição de valores a atributos com uma só letra. Com muita frequência, ao escrever um `__getattr__` você também deverá adicionar `__setattr__` para evitar um comportamento inconsistente.

Implementar a função `__hash__` forneceu o contexto perfeito para usar `functools.reduce`, pois precisávamos aplicar o operador `xor ^` sucessivamente aos hashes de todos os componentes de `Vector` a fim de produzir um valor de hash agregado para todo o `Vector`. Após aplicar `reduce` em `__hash__`, usamos a função de redução embutida `all` para criar um método `__eq__` mais eficiente.

A última melhoria em `Vector` foi reimplementar o método `__format__` de `Vector2d` para suportar coordenadas esféricas como alternativa às coordenadas cartesianas `default`. Usamos muita matemática e vários geradores para implementar `__format__` e suas funções auxiliares, mas são detalhes de implementação – retornaremos aos geradores no capítulo 14. O objetivo da última seção foi dar suporte a uma formatação personalizada e cumprir a promessa de um `Vector` que pode fazer tudo que faz um `Vector2d` e mais um pouco,

Como fizemos no capítulo 9, aqui vimos, com frequência, como os objetos-padrões em Python se comportam para emulá-los e oferecer um comportamento “`pythônico`” a `Vector`.

No capítulo 13, implementaremos vários operadores infixos em `Vector`. A matemática será muito mais simples do que a que usamos no método `angle()`, mas explorar o funcionamento dos operadores infixos em Python é uma ótima lição em design orientado a objetos. Contudo, antes de chegar à sobrecarga de operadores, daremos um passo para trás, deixando de lado o trabalho com uma só classe para ver como organizar várias classes usando interfaces e herança, que serão os assuntos dos capítulos 11 e 12.

Leituras complementares

A maioria dos métodos especiais abordada no exemplo com `Vector` também aparece no exemplo com `Vector2d` no capítulo 9, portanto as referências em “Leituras complementares” na página 313 são todas relevantes nesse caso.

A poderosa função de ordem superior `reduce` também é conhecida como `fold` (dobrar), `accumulate` (acumular), `aggregate` (agregar), `compress` (compactar) e `inject` (injetar). Para obter mais informações, consulte o artigo “Fold (higher-order function)” [Fold (função de ordem superior), [http://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](http://en.wikipedia.org/wiki/Fold_(higher-order_function))] na Wikipedia, que apresenta aplicações dessas funções de ordem superior com ênfase em programação funcional com estruturas de dados recursivas. O artigo inclui também uma tabela que lista funções similares a `fold` em dezenas de linguagens de programação.

Ponto de vista

Protocolos como interfaces informais

Protocolos não são uma invenção de Python. A equipe de Smalltalk, que também cunhou a expressão “orientada a objetos”, usava “protocolo” como sinônimo para o que chamamos hoje de interfaces. Alguns ambientes de programação para Smalltalk permitiam aos programadores marcar um grupo de métodos como um protocolo, mas isso servia somente para ajudar na documentação e na navegação: não era imposto pela linguagem. É por isso que acredito que “interface informal” seja uma explicação breve razoável para “protocolo” quando falo a um público-alvo que tenha mais familiaridade com interfaces formais (verificadas pelo compilador).

Protocolos estabelecidos evoluem naturalmente em qualquer linguagem que use tipagem dinâmica, ou seja, quando a verificação de tipos é feita em tempo de execução por não haver informações de tipagem estática em assinaturas de métodos e em variáveis. Ruby é outra importante linguagem orientada a objetos que tem tipagem dinâmica e usa protocolos.

Na documentação de Python, geralmente, podemos dizer se um protocolo está sendo discutido quando vemos expressões como “um objeto arquivo ou similar” (a *file-like object*). É uma maneira rápida de dizer “algo que se comporta suficientemente como um arquivo, implementando partes da interface de arquivos que sejam relevantes no contexto”.

Você pode achar que implementar apenas parte de um protocolo seja um desleixo, mas tem a vantagem de manter a simplicidade. A seção 3.3 do capítulo “Data Model” (Modelo de dados, <http://bit.ly/pydocs-smn>) sugere o seguinte:

Ao implementar uma classe que emule qualquer tipo embutido, é importante que a emulação seja implementada somente até o ponto em que faça sentido para o objeto modelado. Por exemplo, algumas sequências podem funcionar bem com a recuperação de elementos individuais, mas extrair uma fatia pode não fazer sentido.

— capítulo “Data Model” de *The Python Language Reference*

Quando não precisamos criar métodos sem sentido apenas para respeitar o contrato de uma interface excessivamente complicada e manter o compilador satisfeito, é mais fácil seguir o princípio KISS (http://en.wikipedia.org/wiki/KISS_principle). Terei mais a dizer sobre protocolos e interfaces no capítulo 11, em que eles serão o foco principal.

Origens de duck typing

Acredito que a comunidade Ruby, mais que as outras, ajudou a popularizar o termo “duck typing” (tipagem pato) enquanto discursava para o público Java. Porém a expressão já era usada em discussões sobre Python antes de Ruby e Python serem “populares”. De acordo com a Wikipedia, um dos primeiros exemplos da analogia com pato em programação orientada a objetos está em uma mensagem de Alex Martelli na Python-list em 26 de julho de 2000: *polymorphism (was Re: Type checking in python?)* [polimorfismo: (era Re: verificação de tipos em python?, <http://bit.ly/1QOuTPx>)]. É daí que a citação do início deste capítulo foi extraída. Se estiver curioso sobre as origens literárias do termo “duck typing” e as aplicações desse conceito de orientação a objetos em várias linguagens, dê uma olhada na entrada “Duck typing” na Wikipedia (http://en.wikipedia.org/wiki/Duck_typing).

Um format seguro, com usabilidade melhorada

Ao implementar `__format__`, não tomamos nenhum cuidado em relação a instâncias de `vector` com um número bem elevado de componentes, como fizemos em `__repr__` usando `reprlib`. O raciocínio é que `repr()` serve para depuração e logging, portanto sempre deverá gerar uma saída útil, enquanto `__format__` é usado para exibir uma saída aos usuários finais, que, presumivelmente, querem ver todo o `vector`. Se você acha isso perigoso, seria interessante implementar uma extensão adicional à minilinguagem de especificação de formatação.

Eis como eu faria: por padrão, qualquer `vector` formatado exibiria um número de componentes razoável, porém limitado, por exemplo, 30. Se houver mais elementos além disso, o comportamento-padrão seria semelhante ao de `reprlib`: eliminar o excesso e colocar ... em seu lugar. Entretanto, se o especificador de formatação terminar com o código especial `*` indicando “tudo”, a limitação de tamanho seria desabilitada. Desse modo, um usuário que não estivesse ciente do

problema de apresentações muito longas não seria acidentalmente prejudicado por essa solução. Mas se a limitação default tornar-se um incômodo, a presença de ... deve motivar o usuário a pesquisar a documentação e descobrir o código de formatação *.

Envie um pull request para o repositório de *Fluent Python* no GitHub (<https://github.com/fluentpython/example-code>) se você implementar isso!

A busca de uma soma pythônica

Não há uma única resposta para “O que é pythônico?”, assim como não há uma única resposta para “O que é bonito?”. Dizer que significa usar “Python idiomático”, como faço com frequência, não é 100% satisfatório, pois o que pode ser “idiomático” para você pode não ser para mim. Só sei que “idiomático” não quer dizer usar os recursos mais obscuros da linguagem.

Na Python-list (<https://mail.python.org/mailman/listinfo/python-list>), há uma discussão de abril de 2003 chamada “Pythonic Way to Sum n-th List Element?” (Maneira pythônica de somar o enésimo elemento de uma lista?, <http://bit.ly/1QOv5y5>). Ela é relevante para nossa discussão sobre *reduce* neste capítulo.

Guy Middleton, responsável pelo post original, pediu uma melhoria na solução a seguir, dizendo que não gostava de usar *lambda*:⁷

```
>>> my_list = [[1, 2, 3], [40, 50, 60], [9, 8, 7]]
>>> import functools
>>> functools.reduce(lambda a, b: a+b, [sub[1] for sub in my_list])
60
```

O código usa muitos “idioms”: *lambda*, *reduce* e uma *list comprehension*. Provavelmente, ele ficaria em último lugar em um concurso de popularidade, pois ofende as pessoas que detestam *lambda* e as que desprezam *list comprehensions* – praticamente, os dois lados da polêmica.

Se você vai usar *lambda*, é provável que não haja motivos para usar uma *list comprehension* – exceto para filtrar, o que não é o caso aqui.

Eis uma solução minha que agrada os amantes de *lambda*:

```
>>> functools.reduce(lambda a, b: a + b[1], my_list, 0)
60
```

Não participei da discussão original e não usaria essa solução em um código de verdade porque também não gosto muito de *lambda*, mas queria mostrar um exemplo sem uma *list comprehension*.

⁷ Adapte o código para essa apresentação: em 2003, *reduce* era uma função embutida, mas, em Python 3, precisamos importá-la; além disso, substituí os nomes *x* e *y* por *my_list* e *sub*, para sublista.

A primeira resposta veio de Fernando Perez, criador do IPython, que destacou que NumPy trata arrays n -dimensionais e fatiamentos n -dimensionais:

```
>>> import numpy as np
>>> my_array = np.array(my_list)
>>> np.sum(my_array[:, 1])
60
```

Acho a solução de Perez interessante, mas Guy Middleton elogiou a solução a seguir, de Paul Rubin e Skip Montanaro:

```
>>> import operator
>>> functools.reduce(operator.add, [sub[1] for sub in my_list], 0)
60
```

Então Evan Simpson perguntou: “O que há de errado com isto?”:

```
>>> t = 0
>>> for sub in my_list:
...     total += sub[1]
>>> t
60
```

Muitas pessoas concordaram que era bem pythônico. Alex Martelli chegou a dizer que, provavelmente, é como Guido escreveria o código.

Gosto do código de Evan Simpson, mas gosto também do comentário de David Eppstein sobre ele:

Se quiser a soma de uma lista de itens, escreva o código de maneira que pareça ser “a soma de uma lista de itens”, e não para parecer “um laço para percorrer esses itens, manter outra variável *t*, executar uma sequência de adições”. Por que temos linguagens de alto nível se não for para expressar nossas intenções em um nível mais alto e deixar que a linguagem se preocupe com as operações de baixo nível necessárias para implementá-las?

Então Alex Martelli voltou com uma sugestão:

“A soma” é necessária com tanta frequência que eu não me importaria nem um pouco se Python a implementasse como uma função embutida. Mas “`reduce(operator.add, ...)`” simplesmente não é uma boa maneira de expressá-la, em minha opinião (embora, por ser um antigo programador de APL e gostar de programação funcional, eu *devesse* gostar – mas não gosto).

Alex prossegue sugerindo uma função `sum()`, cuja implementação ele forneceu. Ela se tornou uma função embutida em Python 2.3, lançada somente três meses depois dessa conversa. Desse modo, a sintaxe preferida por Alex tornou-se a norma:

```
>>> sum([sub[1] for sub in my_list])
60
```

No final do ano seguinte (novembro de 2004), Python 2.4 foi lançada com expressões geradoras, oferecendo o que, em minha opinião, é a resposta mais pythônica à pergunta original de Guy Middleton:

```
>>> sum(sub[1] for sub in my_list)
60
```

Essa solução não só é mais legível que `reduce` como também evita a armadilha da sequência vazia: `sum([])` é 0, simples assim.

Na mesma conversa, Alex Martelli sugere que a função embutida `reduce` em Python 2 representava mais problemas do que vantagens, pois incentivava a implementação de códigos difíceis de explicar. Ele foi bem convincente: a função foi rebaixada para o módulo `functools` em Python 3.

Apesar disso, `functools.reduce` tem seu lugar. Ela resolveu o problema de nosso `Vector.__hash__` de uma maneira que eu chamaria de pythônica.

CAPÍTULO 11

Interfaces: de protocolos a ABCs

Uma classe abstrata representa uma interface.¹

– Bjarne Stroustrup
Criador de C++

As interfaces são o assunto deste capítulo: dos protocolos dinâmicos, que são a marca registrada de *duck typing*, às classes-base abstratas (ABCs), que deixam as interfaces explícitas e verificam a conformidade das implementações.

Se você tem experiência anterior com Java, C# ou com uma linguagem semelhante, a novidade aqui está nos protocolos informais de *duck typing*. No entanto, para quem é Pythonista ou Rubyista há muito tempo, essa é a maneira “normal” de pensar nas interfaces, e a novidade é a formalidade e a verificação de tipos das ABCs. A linguagem já tinha 15 anos de idade quando as ABCs foram introduzidas em Python 2.6.

Começaremos recapitulando como a comunidade Python tradicionalmente entende as interfaces como algo flexível – no sentido de que uma interface parcialmente implementada geralmente é aceitável. Deixaremos isso claro por meio de alguns exemplos que enfatizam a natureza dinâmica de *duck typing*.

Em seguida, um artigo de Alex Martelli escrito especialmente para este livro apresentará as ABCs e dará nome a uma nova tendência na programação em Python. O restante do capítulo será dedicado às ABCs, começando pelo seu uso comum como superclasses quando você precisa implementar uma interface. Então veremos quando uma ABC verifica subclasses concretas para saber se estão de acordo com a interface que ela define e como um sistema de registros permite aos desenvolvedores declarar que uma classe implementa uma interface sem usar herança. Por fim, veremos como uma ABC pode ser programada para “reconhecer” automaticamente uma classe qualquer que esteja de acordo com sua interface – sem usar herança ou registros explícitos.

¹ Bjarne Stroustrup, *The Design and Evolution of C++* (Addison-Wesley, 1994), p. 278.

Implementaremos uma nova ABC para ver como ela funciona, mas Alex Martelli e eu não queremos incentivar você a escrever suas próprias ABCs a torto e a direito. O risco de over-engineering (engenharia em excesso) com as ABCs é muito alto.



As ABCs, assim como os descritores e as metaclasses, são ferramentas para criação de frameworks. Sendo assim, apenas uma pequena minoria dos desenvolvedores Python pode criar ABCs sem impor limitações despropositadas e trabalhadeira desnecessária aos colegas programadores.

Vamos começar com a visão pythônica das interfaces.

Interfaces e protocolos na cultura de Python

A linguagem Python já fazia muito sucesso antes de as ABCs terem sido introduzidas, e a maior parte dos códigos existentes não as usa. Desde o capítulo 1, estamos falando de *duck typing* e de protocolos. Na seção “Protocolos e duck typing” na página 322, os protocolos são definidos como as interfaces informais que fazem o polimorfismo funcionar em linguagens com tipagem dinâmica, como Python.

Como as interfaces funcionam em uma linguagem com tipagem dinâmica? Em primeiro lugar, o básico: mesmo sem uma palavra reservada `interface` na linguagem, e independentemente das ABCs, toda classe tem uma interface: os atributos públicos definidos (métodos ou atributos de dados), implementados ou herdados pela classe. Isso inclui métodos especiais como `__getitem__` ou `__add__`.

Por definição, atributos protegidos e privados não fazem parte de uma interface, apesar de “protegido” ser simplesmente uma convenção de nomenclatura (o único underscore na frente) e os atributos privados serem facilmente acessados (lembre-se da seção “Atributos privados e “protegidos” em Python” na página 304). Violar essas convenções não é uma boa ideia.

Por outro lado, não é pecado ter atributos de dados públicos como parte da interface de um objeto, pois – se for necessário – um atributo de dado sempre pode ser transformado em uma propriedade implementando a lógica getter/setter, sem desestruturar o código do cliente que use a sintaxe simples `obj.attr`. Fizemos isso na classe `Vector2d`: no exemplo 11.1, podemos ver a primeira implementação com atributos `x` e `y` públicos.

Exemplo 11.1 – `vector2d_v0.py`: `x` e `y` são atributos de dados públicos (mesmo código do exemplo 9.2)

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.x = float(x)
```

```

        self.y = float(y)
    def __iter__(self):
        return (i for i in (self.x, self.y))
    # há mais métodos a seguir (omitidos desta listagem)

```

No exemplo 9.7, transformamos `x` e `y` em propriedades somente para leitura (Exemplo 11.2). Essa é uma refatoração significativa, porém uma parte essencial da interface de `Vector2d` permaneceu inalterada: os usuários ainda podem ler `my_vector.x` e `my_vector.y`.

Exemplo 11.2 – `vector2d_v3.py`: `x` e `y` reimplementados como propriedades (veja a listagem completa no exemplo 9.9)

```

class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    # há mais métodos a seguir (omitidos desta listagem)

```

Uma definição complementar útil de interface é: o subconjunto dos métodos públicos de um objeto que lhe permitem desempenhar um papel específico no sistema. É isso que está implícito quando a documentação de Python menciona “um objeto arquivo ou similar” (a file-like object) ou “um iterável” sem especificar uma classe. Uma interface vista como um conjunto de métodos para desempenhar um papel é o que os programadores de Smalltalk chamavam de *protocolo*, e o termo foi disseminado em outras comunidades de linguagens dinâmicas. Os protocolos são independentes de herança. Uma classe pode implementar vários protocolos, permitindo que suas instâncias desempenhem diversos papéis.

Protocolos são interfaces, mas, por serem informais – são definidos somente por documentação e convenções –, os protocolos não podem ser verificados estaticamente pelo interpretador como as interfaces formais (veremos como as ABCs impõem a

conformidade com a interface mais adiante neste capítulo). Um protocolo pode ser parcialmente implementado por uma classe em particular, e isso não é um problema. Às vezes, tudo que uma API específica exige de um “objeto arquivo ou similar” é que ele tenha um método `.read()` que devolva bytes. Os demais métodos de arquivo podem ou não ser relevantes no contexto.

Quando escrevi este texto, a documentação de `memoryview` de Python 3 (<http://bit.ly/1QOxU2e>) afirmava que ela funciona com objetos que “ofereçam suporte ao protocolo de buffer”, que está documentado somente no nível da API em C. O construtor `bytearray` (<http://bit.ly/1MDR1Lw>) aceita um “objeto que esteja de acordo com a interface de buffer”. Atualmente, há um movimento para adotar “objeto bytes ou similar” (bytes-like object) como um termo mais amigável.² Destaco essa questão para enfatizar que “objeto X ou similar”, “protocolo X” e “interface X” são sinônimos nas mentes dos Pythonistas.

Uma das interfaces mais importantes em Python é o protocolo de sequência. O interpretador faz de tudo para lidar com objetos que ofereçam uma implementação desse protocolo, por mínima que seja, como mostra a próxima seção.

Python curte seqüências

A filosofia do modelo de dados de Python é cooperar o máximo possível com os protocolos essenciais. Quando se trata de seqüências, Python se esforça bastante para trabalhar até mesmo com as implementações mais simples.

A figura 11.1 mostra como a interface formal `Sequence` é definida como uma ABC.

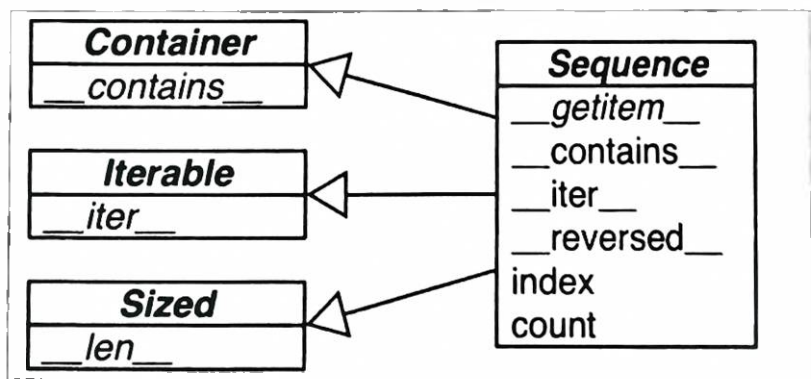


Figura 11.1 – Diagrama de classes UML para a ABC `Sequence` e as classes abstratas relacionadas de `collections.abc`. As setas de herança apontam da subclasse para suas superclasses. Nomes em *itálico* são métodos abstratos.

² Issue 16518: “add *buffer* protocol to glossary” (adicionar protocolo de *buffer* ao glossário, <http://bugs.python.org/issue16518>), na verdade, foi resolvido substituindo várias menções a “object that supports the buffer protocol/interface/API” (objetos que tratam o protocolo /interface/API de *buffer*) por “bytes-like object” (objeto bytes ou similar); um problema relacionado é “Other mentions of the buffer protocol” (Outras menções ao protocolo de *buffer*, <http://bugs.python.org/issue22581>).

Vamos agora dar uma olhada na classe `Foo` do exemplo 11.3. Ela não herda de `abc.Sequence` e implementa apenas um método do protocolo de sequência: `__getitem__` (`__len__` não está presente).

Exemplo 11.3 – Implementação parcial do protocolo de sequência com `__getitem__`: suficiente para acesso a itens, iteração e o operador `in`

```
>>> class Foo:
...     def __getitem__(self, pos):
...         return range(0, 30, 10)[pos]
...
>>> f[1]
10
>>> f = Foo()
>>> for i in f: print(i)
...
0
10
20
>>> 20 in f
True
>>> 15 in f
False
```

Não há um método `__iter__`, mas as instâncias de `Foo` são iteráveis porque – como alternativa –, ao ver um método `__getitem__`, o interpretador Python tenta fazer uma iteração pelo objeto chamando esse método com índices inteiros começando em 0. Como Python é esperto o suficiente para fazer iterações em instâncias de `Foo`, ele também pode fazer o operador `in` funcionar, apesar de `Foo` não ter um método `__contains__`: ele faz uma varredura completa para ver se um item está presente.

Em suma, dada a importância do protocolo de sequência, na ausência de `__iter__` e de `__contains__`, Python ainda consegue fazer a iteração e o operador `in` funcionar chamando `__getitem__`.

Nosso `FrenchDeck` original do capítulo 1 também não é uma subclasse de `abc.Sequence`, mas implementa ambos os métodos do protocolo de sequência: `__getitem__` e `__len__`. Veja o exemplo 11.4.

Exemplo 11.4 – Um baralho como uma sequência de cartas (igual ao exemplo 1.1)

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])
```

```
class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

Boa parte das demos do capítulo 1 funciona por causa do tratamento especial que Python dá a tudo que lembre vagamente uma sequência. A iteração em Python representa uma forma extrema de duck typing: o interpretador tenta usar dois métodos diferentes para iterar pelos objetos.

Vamos agora analisar outro exemplo que enfatiza a natureza dinâmica dos protocolos.

Monkey-patching para implementar um protocolo em tempo de execução

A classe `FrenchDeck` do exemplo 11.4 tem um grande defeito: suas instâncias não podem ser embaralhadas. Anos atrás, quando escrevi o exemplo com `FrenchDeck` pela primeira vez, eu implementara um método `shuffle`. Mais tarde, tive um insight pythônico: se `FrenchDeck` age como uma sequência, ela não precisará ter seu próprio método `shuffle`, pois já existe um `random.shuffle` cuja documentação diz “Shuffle the sequence *x* in place” (Embaralha a sequência *x* in-place, <https://docs.python.org/3/library/random.html#random.shuffle>).



Ao seguir os protocolos estabelecidos, você aumenta suas chances de aproveitar códigos existentes na biblioteca-padrão e de terceiros, graças ao duck typing.

A função `random.shuffle` padrão é usada da seguinte maneira:

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 2, 9, 7, 8, 3, 1, 4, 0, 6]
```

No entanto, se tentarmos embaralhar uma instância de `FrenchDeck`, teremos uma exceção, como no exemplo 11.5.

Exemplo 11.5 – `random.shuffle` não é capaz de tratar `FrenchDeck`

```
>>> from random import shuffle
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> shuffle(deck)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../python3.3/random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

A mensagem de erro é bem clara: “`'FrenchDeck' object does not support item assignment`” (objeto `'FrenchDeck'` não aceita atribuição de itens). O problema é que `shuffle` funciona com base na troca (swap) de itens na coleção, e `FrenchDeck` implementa somente o protocolo de sequência *imutável*. As sequências mutáveis também precisam oferecer um método `__setitem__`.

Como Python é dinâmica, podemos corrigir isso em tempo de execução, até mesmo no console interativo. O exemplo 11.6 mostra como fazer isso.

Exemplo 11.6 – Um monkey patch em `FrenchDeck` para torná-lo mutável e compatível com `random.shuffle` (continuação do exemplo 11.5)

```
>>> def set_card(deck, position, card): ❶
...     deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card ❷
>>> shuffle(deck) ❸
>>> deck[:5]
[Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'), Card(rank='4',
suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9', suit='spades')]
```

- ❶ Cria uma função que aceita `deck`, `position` e `card` como argumentos.
- ❷ Vincula essa função a um atributo chamado `__setitem__` na classe `FrenchDeck`.
- ❸ `deck` agora pode ser embaralhado porque `FrenchDeck` implementa o método necessário do protocolo de sequência mutável.

A assinatura do método especial `__setitem__` está definida em *The Python Language Reference* (Guia de referência à linguagem Python) na seção “3.3.6. Emulating container types” (Emulando tipos para coleção, <http://bit.ly/1QOyDQY>). Nesse caso, chamamos os argumentos de `deck`, `position`, `card` – e não de `self`, `key`, `value`, como no guia de referência

à linguagem – para mostrar que todo método em Python nasce como uma função simples e chamar o primeiro argumento de `self` é apenas uma convenção. Não há problemas em uma sessão de console, mas em um arquivo-fonte Python será muito melhor usar `self`, `key` e `value` conforme documentado.

O truque está em `set_card` saber que o objeto `deck` tem um atributo chamado `_cards`, que precisa ser uma sequência mutável. A função `set_card` é então associada à classe `FrenchDeck` como o método especial `__setitem__`. Esse é um exemplo de *monkey patching*: alterar uma classe ou um módulo em tempo de execução, sem tocar no código-fonte. Monkey patching é poderoso, mas o código que realmente faz o patch é altamente acoplado ao código a ser alterado, frequentemente manipulando partes privadas e não documentadas.

Além de ser um exemplo de monkey patching, o exemplo 11.6 enfatiza que os protocolos são dinâmicos: `random.shuffle` não se importa com o tipo de argumento recebido; ela só precisa que o objeto implemente parte do protocolo de sequência mutável. Não importa nem mesmo se o objeto “nasceu” com os métodos necessários ou se eles foram, de algum modo, adquiridos posteriormente.

O tema deste capítulo até agora foi “duck typing”: trabalhar com objetos independentemente de seus tipos, desde que eles implementem determinados protocolos.

Quando apresentamos diagramas com ABCs, a intenção foi mostrar como os protocolos estão relacionados às interfaces explícitas documentadas nas classes abstratas, mas, na verdade, não herdamos de nenhuma ABC até agora.

Nas próximas seções, vamos aproveitar diretamente as ABCs, e não apenas como documentação.

Aves aquáticas de Alex Martelli

Após analisar as interfaces usuais de Python no estilo de protocolos, vamos passar para as ABCs. Porém, antes de mergulhar em exemplos e detalhes, o convidado Alex Martelli explica em um artigo por que as ABCs foram um ótimo acréscimo em Python.



Sou muito grato a Alex Martelli. Ele já era a pessoa mais citada neste livro antes de ter se tornado um dos revisores técnicos. Suas contribuições foram muito valiosas e ele ainda se ofereceu para escrever esse artigo. Temos muita sorte por tê-lo em nossa comunidade. Agora é com você, Alex!

Aves aquáticas e ABCs

Por Alex Martelli

A Wikipedia me dá o crédito (http://en.wikipedia.org/wiki/Duck_typing#History) por ter ajudado a disseminar o meme útil e expressivo “*duck typing*” (isto é, ignorar o verdadeiro tipo de um objeto e, em vez disso, focar em garantir que o objeto implemente os nomes de métodos, as assinaturas e a semântica exigidos ao uso que se espera dele).

Em Python, isso se reduz, principalmente, a evitar o uso de `isinstance` para verificar o tipo do objeto (sem mencionar a abordagem pior ainda de verificar, por exemplo, se `type(foo) is bar` – que, com razão, é um anátema, pois inibe até mesmo as formas mais simples de herança!)

A abordagem geral de *duck typing* permanece muito útil em vários contextos – apesar disso, em muitos outros, uma opção preferível se desenvolveu com o tempo. E aqui há uma história...

Em gerações recentes, a taxonomia de gêneros e espécies (incluindo a família de patos conhecida como anatídeos, mas sem se limitar a ela) tem sido orientada principalmente pela *fenética* – uma abordagem focada em semelhanças de morfologia e de comportamento... acima de tudo, traços *observáveis*. A analogia com “*duck typing*” era forte.

Entretanto a evolução paralela frequentemente pode produzir traços semelhantes, tanto morfológicos quanto comportamentais, entre espécies que, na verdade, não estão relacionadas, mas que, por acaso, evoluíram em nichos ecológicos similares, embora separados. “Semelhanças acidentais” similares também acontecem em programação – por exemplo, considere o exemplo clássico de POO³:

```
class Artist:
    def draw(self): ...

class Gunslinger:
    def draw(self): ...

class Lottery:
    def draw(self): ...
```

3 Nota do autor/revisor: este exemplo clássico não faz sentido em português, mas merece uma explicação. Ele se baseia em três significados diferentes do verbo “to draw”. A primeira classe representa um artista, e o método “draw” seria traduzido como desenhar. A segunda classe representa um pistoleiro, e o método “draw” é sacar (isto é, sacar uma arma do coldre). A última classe é uma loteria, e “draw” seria tirar ou sortear um número.

É óbvio que a simples existência de um método chamado `draw`, que pode ser chamado sem argumentos, está longe de ser suficiente para nos garantir que dois objetos `x` e `y` tais que `x.draw()` e `y.draw()` possam ser chamados, sejam, de algum modo, intercambiáveis ou abstratamente equivalentes – nada pode ser inferido sobre a similaridade da semântica resultante dessas chamadas. Precisamos de um programador com conhecimentos que *assegure* que tal equivalência existe em algum nível! Em biologia (e em outras disciplinas), esse problema levou ao surgimento (e em muitos aspectos, à predominância) de uma abordagem alternativa à fenética, conhecida como *cladística* – que foca as decisões taxonômicas em características herdadas de ancestrais comuns, em vez daquelas que evoluíram de modo independente. (O sequenciamento simples e rápido de DNA pode deixar a cladística extremamente prática em muitos casos, nos últimos anos.)

Por exemplo, os *sheldgeese*⁴ (antes classificados como mais próximos de outros gansos) e os patos-brancos (antes classificados como mais próximos dos patos) atualmente estão agrupados na subfamília *Tadornidae* (implicando que eles são mais semelhantes entre si do que qualquer outro anatídeo, pois compartilham um ancestral comum mais próximo). Além do mais, a análise de DNA mostrou, em particular, que o pato-de-asas-brancas não é tão próximo do pato-do-mato (esse último, um pato-branco) como a semelhança na aparência e no comportamento sugeriram por muito tempo – desse modo, o pato-de-asas-brancas foi reclassificado em seu próprio gênero, saindo totalmente da subfamília!

Isso é importante? Depende do contexto! Para propósitos como decidir qual é a melhor maneira de preparar uma ave aquática depois de tê-la caçado, por exemplo, traços observáveis específicos (nem todos – a plumagem, por exemplo, é da menor importância nesse contexto), principalmente textura e sabor (boa e velha fenética!), podem ser muito mais relevantes do que a cladística. Mas, para outras questões, como suscetibilidade a diferentes agentes patogênicos (se você estiver tentando criar aves aquáticas em cativeiro ou preservá-las em seu ambiente), a semelhança de DNA pode ser muito mais importante...

Sendo assim, usando uma analogia bem flexível com essas revoluções taxonômicas no mundo das aves aquáticas, recomendo suplementar (e não substituir totalmente – em determinados contextos, ele ainda poderá servir) o bom e velho *duck typing* (tipagem pato) por... *goose typing* (tipagem ganso)!

O significado de *goose typing* é: `isinstance(obj, cls)` agora não tem problemas... desde que `cls` seja uma classe-base abstrata – em outras palavras, a metaclasses de `cls` é `abc.ABCMeta`.

⁴ N.T.: Aves da família dos anatídeos do gênero *Chloephaga*. Entre as espécies dessa família estão o ganso-do-campo (*Chloephaga picta*) e o ganso-de-cabeça-ruiva (*Chloephaga rubidiceps*).

Você pode encontrar muitas classes abstratas úteis em `collections.abc` (e outras no módulo `numbers` em *The Python Standard Library*).⁵

Entre as diversas vantagens conceituais das ABCs em relação às classes concretas [por exemplo, a recomendação de que “todas as classes que não são folhas devem ser abstratas” de Scott Meyer – veja o Item 33 (<http://ptgmedia.pearsoncmg.com/images/O20163371x/items/item33.html>) de seu livro *More Effective C++*], as ABCs de Python acrescentam uma vantagem prática importante: o método de classe `register`, que permite ao código do usuário final “declarar” que uma determinada classe se torne uma subclasse “virtual” de uma ABC (para isso, a classe registrada deve atender aos requisitos da ABC para nomes e assinaturas de métodos e, acima de tudo, ao contrato semântico subjacente – mas não precisa ter sido desenvolvida com qualquer conhecimento da ABC e, em particular, não precisa herdar dela!). Isso contribui bastante para quebrar a rigidez e o forte acoplamento que faz da herança algo a ser usado com muito mais cuidado do que normalmente é feito pela maioria dos programadores que usam POO...

Às vezes, você nem precisará registrar uma classe para que uma ABC a reconheça como uma subclasse!

É isso que ocorre com as ABCs cuja essência se reduza a alguns métodos especiais. Por exemplo:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
```

Como você pode ver, `abc.Sized` reconhece `Struggle` como “uma subclasse”, sem a necessidade de registro, pois implementar o método especial `__len__` é tudo que é necessário (presume-se que ele seja implementado com a sintaxe e a semântica apropriadas, ou seja, invocável sem argumentos e devolvendo um inteiro não negativo que indica o “tamanho” do objeto, respectivamente; qualquer código que implemente um método de nome especial, por exemplo, `__len__`, com uma sintaxe ou uma semântica qualquer, que não estejam de acordo com isso, terá problemas muito piores, de qualquer modo).

⁵ É claro que você também pode definir suas próprias ABCs – mas não incentivo ninguém, exceto os Pythonistas mais experientes, a seguir por esse caminho, assim como não os incentivaria a definir suas próprias metaclasses... e até para os ditos “Pythonistas mais experientes”, aqueles entre nós que apresentam um profundo domínio dos mínimos detalhes da linguagem, essas não são ferramentas para uso frequente: essa “metaprogramação profunda”, se for apropriada, é voltada para autores de frameworks amplos, projetados para serem estendidos de modo independente por um grande número de equipes diferentes de desenvolvimento... menos de 1% dos “Pythonistas mais experientes” precisará disso! — A.M.

Então eis a minha recomendação final: sempre que você implementar uma classe que englobe qualquer um dos conceitos representados nas ABCs em `numbers`, `collections.abc` ou em outro framework que você possa estar usando, certifique-se (se for necessário) de criar uma subclasse ou registrá-la na ABC correspondente. No início de seus projetos que usem alguma biblioteca ou framework com classes que tenham deixado de fazer isso, faça você mesmo os registros; em seguida, quando precisar conferir se um argumento (o mais comum) é, por exemplo, “uma sequência”, use:

```
isinstance(the_arg, collections.abc.Sequence)
```

E *não* defina ABCs próprias (ou metaclasses) em código de produção... se sentir vontade de fazer isso, aposto que será um caso de síndrome de “todos os problemas se parecem com um prego” para alguém que acabou de ganhar um martelo novinho em folha – você e os futuros mantenedores de seu código serão mais felizes atendo-se a códigos descomplicados e simples, evitando mergulhar nessas profundezas. *Valê!*

Além de cunhar o termo “goose typing” (tipagem-ganso), Alex destaca que herdar de uma ABC é mais que implementar os métodos necessários: é também uma declaração clara da intenção do desenvolvedor. Essa intenção também pode ser explicitada pelo registro de uma subclasse virtual.

Além disso, o uso de `isinstance` e `issubclass` torna-se mais aceitável para testes em relação às ABCs. No passado, essas funções trabalhavam contra o duck typing, mas com ABCs, elas se tornaram mais flexíveis. Afinal de contas, se um componente não implementar uma ABC por herança, ele sempre poderá ser registrado no futuro para que passe pelas verificações explícitas de tipo.

No entanto, mesmo com ABCs, você deve estar ciente de que o uso excessivo de verificações com `isinstance` pode ser um *code smell* (código que não cheira bem) – um sintoma de design orientado a objetos ruim. Geralmente, ter uma cadeia de `if/elif/elif` com verificações `isinstance` realizando ações diferentes de acordo com o tipo de um objeto é um problema: você deveria estar usando polimorfismo para isso – ou seja, fazendo o design de suas classes de modo que o interpretador faça chamadas aos métodos apropriados, em vez de deixar a lógica de dispatch (despacho) fixa em blocos `if/elif/elif`.



Há uma exceção comum e prática à recomendação anterior: algumas APIs de Python aceitam um único `str` ou uma sequência de itens `str`; se for um único `str`, você vai querer inseri-lo em uma `list` para facilitar o processamento. Como `str` é um tipo de sequência, a maneira mais simples de distingui-lo de outras sequências imutáveis é fazer uma verificação explícita com `isinstance(x, str)`.^(*)

- (*) Infelizmente, em Python 3.4, não há uma ABC que ajude a distinguir um `str` de um `tuple` ou de outras sequências imutáveis, portanto devemos testar em relação a `str`. Em Python 2, existe o tipo `basestr` que ajuda em testes desse tipo. Ele não é uma ABC, mas é uma superclasse tanto de `str` quanto de `unicode`; no entanto, em Python 3, `basestr` não existe mais. Curiosamente, em Python 3, há um tipo `collections.abc.ByteString`, mas ele só ajuda a detectar `bytes` e `bytearray`.

Por outro lado, normalmente, não há problemas em realizar uma verificação com `isinstance` em relação a uma ABC se você precisa impor um contrato de API: “Cara, você precisa implementar isso se quiser me chamar”, como escreveu o revisor técnico Lennart Regebro. Isso é particularmente útil em sistemas que tenham uma arquitetura de plug-ins. Fora do contexto de frameworks, duck typing geralmente é mais simples e mais flexível que verificação de tipos e goose typing.

Por exemplo, em várias classes deste livro, quando precisei pegar uma sequência de itens e processá-los como uma `list`, em vez de exigir um argumento `list` fazendo verificação de tipo, simplesmente aceitei o argumento e criei imediatamente uma `list` a partir dele: dessa maneira, posso aceitar qualquer iterável e, se o argumento não for iterável, a chamada falhará rapidamente com uma mensagem bem clara. Um exemplo desse padrão de código está no método `__init__` do exemplo 11.13, mais adiante neste capítulo. É claro que essa abordagem não funciona se o argumento de sequência não puder ser copiado, seja porque é grande demais ou porque meu código precisa modificá-lo in-place. Nesse caso, um `isinstance(x, abc.MutableSequence)` é melhor. Se qualquer iterável for aceitável, chamar `iter(x)` para obter um iterador é a solução, como veremos na seção “Por que sequências são iteráveis: a função `iter`” na página 453.

Outro exemplo é imitar o tratamento do argumento `field_names` em `collections.namedtuple` (<https://docs.python.org/3/library/collections.html#collections.namedtuple>): `field_names` aceita uma única string com identificadores separados por espaços ou vírgulas, ou uma sequência de identificadores. Pode ser tentador usar `isinstance`, mas o exemplo 11.7 mostra como eu faria isso usando duck typing.⁶

Exemplo 11.7 – Duck typing para tratar uma string ou um iterável de strings

try: ❶

`field_names = field_names.replace(',', ' ').split()` ❷

except `AttributeError`: ❸

⁶ Esse trecho de código foi extraído do exemplo 21.2.

```
pass ❹
field_names = tuple(field_names) ❺
```

- ❶ Supõe que é uma string (EAFP = it's easier to ask forgiveness than permission, ou seja, é mais fácil pedir perdão que permissão).
- ❷ Converte vírgulas em espaços e quebra o resultado em uma lista de nomes.
- ❸ Perdão, `field_names` não faz quack como um `str...` não há `.replace` ou ele devolve algo em que `.split` não pode ser aplicado.
- ❹ Agora supomos que já seja um iterável de nomes.
- ❺ Para ter certeza de que é um iterável e para manter nossa própria cópia, criamos uma tupla a partir do que temos.

Por fim, em seu artigo, Alex enfatiza mais de uma vez a necessidade de moderação na criação de ABCs. Uma epidemia de ABCs seria desastrosa, impondo um cerimonial excessivo em uma linguagem que se tornou popular por ser prática e pragmática. Durante o processo de revisão de *Python fluente*, Alex escreveu o seguinte:

As ABCs foram criadas para encapsular conceitos muito genéricos e abstrações introduzidos por um framework – ideias como “uma sequência” e “um número exato”. É provável que os leitores não precisem escrever novas ABCs; basta usar corretamente as que existem para obter 99,9% dos benefícios, sem sérios riscos de designs equivocados.

Vamos agora ver o *goose typing* na prática.

Criando subclasses de uma ABC

Seguindo o conselho de Martelli, vamos aproveitar uma ABC existente, `collections.MutableSequence`, antes de nos atrevermos a inventar nossa própria ABC. No exemplo 11.8, `FrenchDeck2` é declarada explicitamente como uma subclasse de `collections.MutableSequence`.

Exemplo 11.8 – `frenchdeck2.py`: `FrenchDeck2`, uma subclasse de `collections.MutableSequence`

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck2(collections.MutableSequence):
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]
```

```
def __len__(self):
    return len(self._cards)

def __getitem__(self, position):
    return self._cards[position]

def __setitem__(self, position, value): # ❶
    self._cards[position] = value

def __delitem__(self, position): # ❷
    del self._cards[position]

def insert(self, position, value): # ❸
    self._cards.insert(position, value)
```

- ❶ `__setitem__` é tudo de que precisamos para poder embaralhar...
- ❷ Mas ser uma subclasse de `MutableSequence` nos força a implementar `__delitem__`, que é um método abstrato dessa ABC.
- ❸ Também precisamos implementar `insert`, que é o terceiro método abstrato de `MutableSequence`.

Python não verifica a implementação dos métodos abstratos em tempo de importação (quando o módulo *frenchdeck2.py* é carregado e compilado), mas somente em tempo de execução, quando realmente tentamos instanciar `FrenchDeck2`. Então, se deixarmos de implementar algum método abstrato, teremos uma exceção `TypeError` com uma mensagem como "Can't instantiate abstract class `FrenchDeck2` with abstract methods `__delitem__`, `insert`" (Não é possível instanciar a classe abstrata `FrenchDeck2` com métodos abstratos `__delitem__`, `insert`). É por isso que devemos implementar `__delitem__` e `insert`, mesmo que nossos exemplos com `FrenchDeck2` não precisem desses comportamentos: a ABC `MutableSequence` os exige.

Como mostra a figura 11.2, nem todos os métodos das ABCs `Sequence` e `MutableSequence` são abstratos.

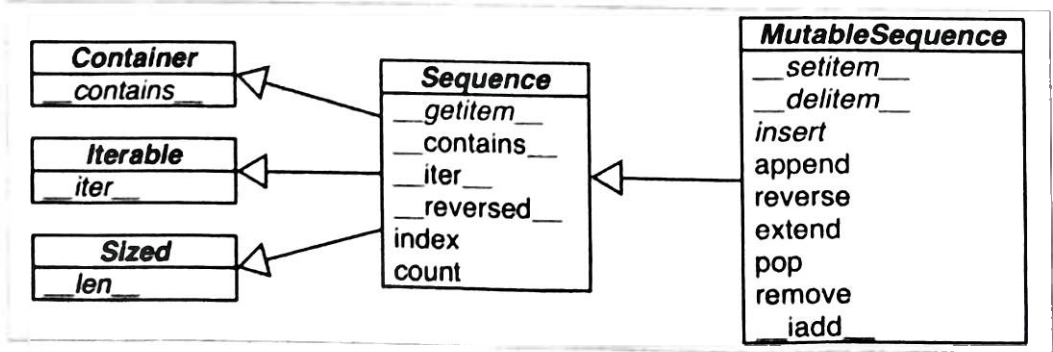


Figura 11.2 – Diagrama UML de classes para a ABC `MutableMapping` e suas superclasses de *collections.abc* (as setas de herança apontam das subclasses para os ancestrais; os nomes em itálico são classes e métodos abstratos).

FrenchDeck2 herda de Sequence os seguintes métodos concretos prontos para usar: `__contains__`, `__iter__`, `__reversed__`, `index` e `count`. De `MutableSequence`, ela recebe `append`, `reverse`, `extend`, `pop`, `remove` e `__iadd__`.

Os métodos concretos de cada ABC em `collections.abc` são implementados em termos da interface pública da classe, portanto funcionam sem conhecimento algum da estrutura interna das instâncias.



Como criador de uma subclasse concreta, você poderá sobrescrever os métodos herdados de ABCs com implementações mais eficientes. Por exemplo, `__contains__` funciona percorrendo a sequência toda, mas se sua sequência concreta mantiver os itens ordenados, você pode escrever um `__contains__` mais rápido, que faça uma busca binária usando a função `bisect` (veja a seção “Administrando sequências ordenadas com `bisect`” na página 71).

Para fazer bom uso das ABCs, você precisa saber o que está disponível. Analisaremos as ABCs de coleções a seguir.

ABCs da biblioteca-padrão

Desde Python 2.6, as ABCs estão disponíveis na biblioteca-padrão. A maioria está definida no módulo `collections.abc`, mas há outras. Você pode encontrar ABCs nos pacotes `numbers` e `io`, por exemplo. Mas o módulo mais amplamente usado é `collections.abc`. Vamos ver o que está disponível nele.

ABCs em `collections.abc`



Há dois módulos de nome `abc` na biblioteca-padrão. Nesse caso, estamos falando de `collections.abc`. Para reduzir o tempo de carga, em Python 3.4, ele está implementado fora do pacote `collections`, em `Lib/_collections_abc.py` (<http://bit.ly/1QOA3Lt>), portanto é importado separadamente de `collections`. O outro módulo `abc` é apenas `abc` (ou seja, `Lib/abc.py` – <https://hg.python.org/cpython/file/3.4/Lib/abc.py>), em que a classe `abc.ABC` está definida. Toda ABC depende dela, mas não precisamos importá-la por conta própria, exceto para criar uma nova ABC.

A figura 11.3 é um diagrama de classes UML resumido (sem nomes de atributos) de todas as 16 ABCs em `collections.abc` em Python 3.4. A documentação oficial de `collections.abc` tem uma bela tabela (<http://bit.ly/1QOA9T8>) que sintetiza as ABCs, seus relacionamentos e seus métodos abstratos e concretos (chamados “métodos mixin”). Há muita herança múltipla presente na figura 11.3. Dedicaremos a maior parte do

capítulo 12 à herança múltipla, mas por enquanto é suficiente dizer que, normalmente, ela não é um problema no que diz respeito às ABCs.⁷

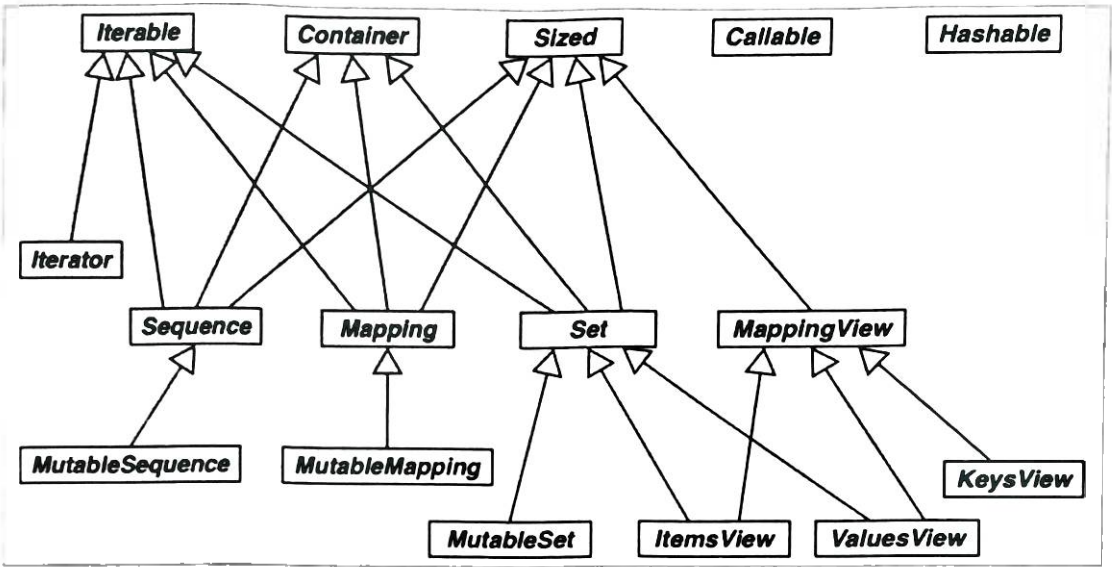


Figura 11.3 – Diagrama de classes UML para ABCs em *collections.abc*.

Vamos analisar os agrupamentos de classes da figura 11.3:

Iterable, Container e Sized

Toda coleção deve herdar dessas ABCs ou, no mínimo, implementar protocolos compatíveis. *Iterable* suporta iteração com `__iter__`, *Container* suporta o operador `in` com `__contains__` e *Sized* suporta `len()` com `__len__`.

Sequence, Mapping e Set

Esses são os tipos principais de coleções imutáveis, e cada um tem uma subclasse mutável. Um diagrama detalhado de *MutableSequence* está na figura 11.2; para *MutableMapping* e *MutableSet*, há diagramas no capítulo 3 (Figuras 3.1 e 3.2).

MappingView

Em Python 3, os objetos devolvidos pelos métodos de mapeamento `.items()`, `.keys()` e `.values()` herdam de *ItemsView*, *KeysView* e *ItemsView*, respectivamente. Os dois primeiros também herdam a rica interface de *Set*, com todos os operadores que vimos na seção “Operações de conjuntos” na página 113.

Callable e Hashable

Essas ABCs não estão intimamente relacionadas com as coleções, mas *collections.abc* foi o primeiro pacote a definir ABCs na biblioteca-padrão, e essas duas foram consideradas importantes o suficiente para serem incluídas. Nunca vi subclasses

⁷ A herança múltipla foi considerada prejudicial e excluída de Java, exceto para interfaces: as interfaces Java podem estender várias interfaces, e as classes Java podem implementar várias interfaces.

de `Callable` nem de `Hashable`. Seu uso principal é oferecer suporte à função embutida `isinstance` como uma maneira segura de determinar se um objeto é invocável (`callable`) ou hashable.⁸

Iterator

Observe que um iterador é uma subclasse de `Iterable`. Discutiremos mais esse assunto no capítulo 14.

Depois do pacote `collections.abc`, o pacote mais útil de ABCs na biblioteca-padrão é `numbers`, que será discutido a seguir.

A torre numérica de ABCs

O pacote `numbers` (<https://docs.python.org/3/library/numbers.html>) define a chamada “torre numérica” (ou seja, a hierarquia linear de ABCs a seguir), em que `Number` é a superclasse de mais alto nível, `Complex` é a subclasse logo depois e assim por diante, até `Integral`:

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

Portanto, se você precisar verificar um inteiro, use `isinstance(x, numbers.Integral)` para aceitar `int`, `bool` (que é subclasse de `int`) ou outros tipos inteiros oferecidos por bibliotecas externas que registrem seus tipos com as ABCs de `numbers`. Para satisfazer sua verificação, você ou os usuários de sua API sempre poderão registrar qualquer tipo compatível como uma subclasse virtual de `numbers.Integral`.

Por outro lado, se um valor puder ser um tipo de ponto flutuante, escreva `isinstance(x, numbers.Real)`, e seu código aceitará `bool`, `int`, `float`, `fractions.Fraction` ou qualquer outro tipo numérico não complexo oferecido por uma biblioteca externa, como a `NumPy`, que esteja devidamente registrado.



`decimal.Decimal` não está registrado como uma subclasse virtual de `numbers.Real`, o que, de certo modo, é surpreendente. O motivo para isso é que, se houver necessidade da precisão de `Decimal` em seu programa, você vai querer estar protegido da mistura accidental de decimais com outros tipos numéricos menos precisos, particularmente, os números de ponto flutuante.

⁸ Para detectar invocáveis, existe a função embutida `callable()` – mas não há uma função `hashable()` equivalente, portanto `isinstance(my_obj, Hashable)` é a maneira preferível de testar se um objeto é hashable.

Depois de ver algumas ABCs existentes, vamos praticar o *goose typing* implementando uma ABC do zero e colocando-a em uso. O objetivo aqui não é incentivar todo mundo a sair por aí criando ABCs, mas aprender a ler o código-fonte das ABCs que você encontrará na biblioteca-padrão e em outros pacotes.

Definindo e usando uma ABC

Para justificar a criação de uma ABC, precisamos ter um contexto para usá-la como um ponto de extensão em um framework. Eis o nosso contexto: suponha que você precise exibir propagandas em um site ou em um aplicativo móvel em ordem aleatória, mas sem repetir as propagandas antes que o conjunto completo tenha sido mostrado. Vamos agora supor que estejamos criando um framework para gerenciamento de propagandas chamado ADAM. Um dos requisitos é aceitar classes definidas pelo usuário que façam seleção aleatória, sem repetição.⁹ Para deixar claro aos usuários de ADAM o que se espera de um componente que faça “seleção aleatória, sem repetição”, definiremos uma ABC.

Inspirado em “pilha” e “fila” (que descrevem interfaces abstratas em termos de organizações físicas de objetos), usarei uma metáfora do mundo real para dar nome à nossa ABC: gaiolas de bingo e máquinas para sortear números de loteria foram projetadas para selecionar itens aleatoriamente a partir de um conjunto finito, sem repetição, até que o conjunto se esgote.

A ABC se chamará *Tombola*, por causa do nome italiano para bingo e do contêiner giratório que mistura os números.¹⁰

A ABC *Tombola* tem quatro métodos. Os dois métodos abstratos são:

- `.load(...)`: coloca itens na coleção.
- `.pick()`: remove um item aleatoriamente da coleção, devolvendo-o.

Os métodos concretos são:

- `.loaded()`: devolve `True` se houver pelo menos um item na coleção.
- `.inspect()`: devolve um `tuple` ordenado, criado a partir dos itens presentes no momento na coleção, sem alterar seu conteúdo (sua ordem interna não é preservada).

A figura 11.4 mostra a ABC *Tombola* e três implementações concretas.

9 Talvez o cliente precise auditar o gerador de números aleatórios ou a agência queira fornecer um gerador viciado. Nunca se sabe...

10 O Oxford English Dictionary define *tombola* como “A kind of lottery resembling lotto” (um tipo de loteria que se parece com bingo).

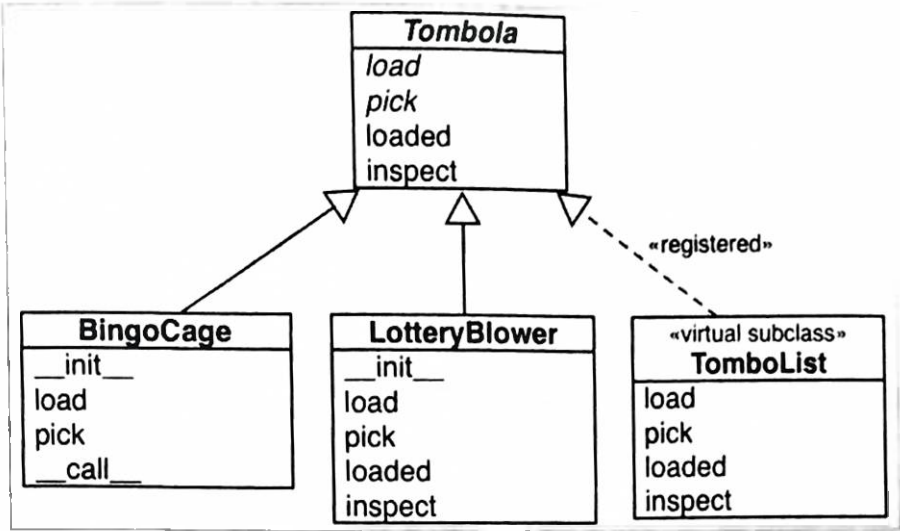


Figura 11.4 – Diagrama UML de uma ABC e três subclasses. O nome da ABC *Tombola* e seus métodos abstratos estão em *itálico*, de acordo com as convenções de UML. A seta tracejada é usada para implementação de interface; nesse caso, ela é usada para mostrar que *TomboList* é uma subclasse virtual de *Tombola* porque ela está registrada, como veremos mais adiante neste capítulo.¹¹

O exemplo 11.9 mostra a definição da ABC *Tombola*.

Exemplo 11.9 – *tombola.py*: *Tombola* é uma ABC com dois métodos abstratos e dois métodos concretos

```
import abc

class Tombola(abc.ABC): ❶

    @abc.abstractmethod
    def load(self, iterable): ❷
        """Adiciona itens a partir de um iterável."""

    @abc.abstractmethod
    def pick(self): ❸
        """Remove um item aleatoriamente, devolvendo-o.

        Esse método deve levantar `LookupError` quando a instância estiver vazia.
        """

    def loaded(self): ❹
        """Devolve `True` se houver pelo menos um item, `False` caso contrário."""
        return bool(self.inspect()) ❺

    def inspect(self):
        """Devolve uma tupla ordenada com os itens contidos no momento"""
        items = []
        while True: ❻
            try:
                items.append(self.pick())
```

¹¹ «registered» (registrada) e «virtual subclass» (subclasse virtual) não são termos-padrões de UML. Nós os usamos para representar um relacionamento de classes específico de Python.

```

    except LookupError:
        break
    self.load(items) ❷
    return tuple(sorted(items))

```

- ❶ Para definir uma ABC, crie uma subclasse de `abc.ABC`.
- ❷ Um método abstrato é marcado com o decorador `@abstractmethod`, e geralmente seu corpo estará vazio, exceto por uma docstring.¹²
- ❸ A docstring instrui os implementadores a levantar `LookupError` se não houver itens para selecionar.
- ❹ Uma ABC pode incluir métodos concretos.
- ❺ Os métodos concretos em uma ABC devem contar somente com a interface definida pela ABC (isto é, outros métodos concretos ou abstratos ou propriedades da ABC).
- ❻ Não sabemos como as subclasses concretas armazenarão os itens, mas podemos criar o resultado de `inspect` esvaziando a `Tombola` com chamadas sucessivas a `.pick()`...
- ❼ ... em seguida, usar `.load(...)` para colocar tudo de volta.



Um método abstrato, na verdade, pode ter uma implementação. Mesmo que tiver, as subclasses continuarão sendo forçadas a sobrescrevê-lo, mas poderão chamar o método abstrato com `super()`, acrescentando funcionalidades a ele em vez de implementá-lo do zero. Consulte a documentação do módulo `abc` (<https://docs.python.org/3/library/abc.html>) para ver detalhes sobre o uso de `@abstractmethod`.

O método `.inspect()` do exemplo 11.9 talvez seja um exemplo tolo, mas mostra que, dados `.pick()` e `.load(...)`, podemos inspecionar o que está na `Tombola` tirando todos os itens e carregando-os de volta. O ponto principal nesse exemplo é enfatizar que não há problemas em oferecer métodos concretos em ABCs, desde que eles dependam somente de outros métodos da interface. Por ter conhecimento de suas estruturas de dados internas, as subclasses concretas de `Tombola` sempre podem sobrescrever `.inspect()` com uma implementação mais esperta, mas não precisam fazê-lo.

O método `.loaded()` do exemplo 11.9 pode não parecer tolo, mas é muito custoso: ele chama `.inspect()` para construir o `tuple` ordenado somente para `lhe aplicar bool()`. Isso funciona, mas uma subclasse concreta pode fazer algo muito melhor, como veremos.

Observe que nossa implementação trabalhosa de `.inspect()` exige que capturemos um `LookupError` lançado por `self.pick()`. O fato de `self.pick()` poder levantar um `LookupError` também faz parte de sua interface, mas não há maneiras de declarar isso

¹² Antes de as ABCs existirem, os métodos abstratos usavam o comando `raise NotImplementedError` para indicar que as subclasses eram responsáveis por sua implementação.

em Python, exceto na documentação (veja a docstring para o método abstrato `pick` no exemplo 11.9).

Escolhi a exceção `LookupError` por causa de seu lugar na hierarquia de exceções de Python em relação a `IndexError` e `KeyError`, as exceções mais prováveis de serem levantadas pelas estruturas de dados usadas na implementação de uma tómbola concreta. Desse modo, as implementações podem levantar `LookupError`, `IndexError` ou `KeyError` para estarem de acordo com essa interface `Tombola`. Veja o exemplo 11.10 (para uma árvore completa, consulte a seção “5.4. Exception hierarchy” [Hierarquia de exceções] de *The Python Standard Library*).

Exemplo 11.10 – Parte da hierarquia de classes de Exception

```
BaseException
├── SystemExit
├── KeyboardInterrupt
├── GeneratorExit
├── Exception
│   ├── StopIteration
│   ├── ArithmeticError
│   │   ├── FloatingPointError
│   │   ├── OverflowError
│   │   └── ZeroDivisionError
│   ├── AssertionError
│   ├── AttributeError
│   ├── BufferError
│   ├── EOFError
│   ├── ImportError
│   ├── LookupError ❶
│   │   ├── IndexError ❷
│   │   └── KeyError ❸
│   ├── MemoryError
│   ... etc.
```

- ❶ `LookupError` é a exceção que tratamos em `Tombola.inspect`.
- ❷ `IndexError` é a subclasse de `LookupError` levantada quando tentamos obter um item de uma sequência com um índice além da última posição.
- ❸ `KeyError` é levantada quando usamos uma chave inexistente para obter um item de um mapeamento.

Agora temos nossa própria ABC `Tombola`. Para observar a verificação de interface realizada por uma ABC, vamos tentar enganar `Tombola` com uma implementação defeituosa no exemplo 11.11.

Exemplo 11.11 – Uma Tombola falsa não passa despercebida

```
>>> from tombola import Tombola
>>> class Fake(Tombola): # ❶
...     def pick(self):
...         return 13
...
>>> Fake # ❷
<class '__main__.Fake'>
<class 'abc.ABC'>, <class 'object'>)
>>> f = Fake() # ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract methods load
```

- ❶ Declara Fake como uma subclasse de Tombola.
- ❷ A classe foi criada sem erros, até agora.
- ❸ TypeError é levantada quando tentamos instanciar Fake. A mensagem é bem clara: Fake é considerada abstrata porque não implementou load, que é um dos métodos abstratos declarados na ABC Tombola.

Assim, temos nossa primeira ABC definida, e nós a colocamos para funcionar validando uma classe. Em breve, vamos criar uma subclasse da ABC Tombola, mas, antes disso, precisamos discutir algumas regras de criação de ABCs.

Detalhes de sintaxe das ABCs

A melhor maneira de declarar uma ABC é criar uma subclasse de `abc.ABC` ou de outra ABC.

No entanto a classe `abc.ABC` é nova em Python 3.4, portanto, se você usa uma versão mais antiga de Python – e acha que não faz sentido criar uma subclasse de outra ABC existente –, use a palavra reservada `metaclass=` no comando `class`, apontando-a para `abc.ABCMeta` (e não para `abc.ABC`). No exemplo 11.9, escreveríamos:

```
class Tombola(metaclass=abc.ABCMeta):
    # ...
```

O argumento nomeado `metaclass=` foi introduzido em Python 3. Em Python 2, use o atributo de classe `__metaclass__`:

```
class Tombola(object): # isso é Python 2!!!
    __metaclass__ = abc.ABCMeta
    # ...
```

Explicaremos as metaclasses no capítulo 21. Por enquanto, vamos aceitar que uma metaclasses é um tipo especial de classe e combinar que uma ABC é um tipo especial de classe; por exemplo, classes “normais” não verificam subclasses, portanto esse é um comportamento especial das ABCs.

Além de `@abstractmethod`, o módulo `abc` define os decoradores `@abstractclassmethod`, `@abstractstaticmethod` e `@abstractproperty`. Contudo esses três últimos se tornaram obsoletos em Python 3.3, quando se tornou possível empilhar decoradores sobre `@abstractmethod`, o que deixou os demais redundantes. Por exemplo, a maneira preferível de declarar um método de classe abstrato é:

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```



A ordem dos decoradores de função empilhados normalmente é importante e, no caso de `@abstractmethod`, a documentação é explícita:

Quando `abstractmethod()` é aplicado em conjunto com outros descritores de método, ele deve ser aplicado como o decorador mais interno... (*)

Em outras palavras, nenhum outro decorador poderá aparecer entre `@abstractmethod` e o comando `def`.

(*) Entrada `@abc.abstractmethod` (<http://bit.ly/1QOFpGB>) na documentação do módulo `abc` (<https://docs.python.org/dev/library/abc.html>).

Agora que já discutimos todas essas questões da sintaxe de ABCs, vamos usar `Tombola` implementando alguns descendentes concretos e funcionalmente completos dessa ABC.

Herdando da ABC `Tombola`

Dada a ABC `Tombola`, vamos agora desenvolver duas subclasses concretas que satisfaçam a sua interface. Essas classes estão representadas na figura 11.4, juntamente com a subclasse virtual que será discutida na próxima seção.

A classe `BingoCage` no exemplo 11.12 é uma variação do exemplo 5.8 usando um gerador de aleatoriedade melhor. Este `BingoCage` implementa os métodos abstratos necessários `load` e `pick`, herda `loaded` de `Tombola`, sobrescreve `inspect` e acrescenta `__call__`.

Exemplo 11.12 – bingo.py: BingoCage é uma subclasse concreta de Tombola

```
import random

from tombola import Tombola

class BingoCage(Tombola): ❶
    def __init__(self, items):
        self._randomizer = random.SystemRandom() ❷
        self._items = []
        self.load(items) ❸

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items) ❹

    def pick(self): ❺
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self): ❻
        self.pick()
```

- ❶ Essa classe BingoCage estende explicitamente Tombola.
- ❷ Finja que usaremos isso em jogos online que envolvem dinheiro. `random.SystemRandom` implementa a API `random` com base na função `os.urandom(...)`, que oferece bytes aleatórios “adequados para uso em criptografia”, de acordo com a documentação do módulo `os` (<http://docs.python.org/3/library/os.html#os.urandom>).
- ❸ Delega a carga inicial ao método `.load(...)`.
- ❹ Em vez da função simples `random.shuffle()`, usamos o método `.shuffle()` de nossa instância de `SystemRandom`.
- ❺ `pick` é implementado como no exemplo 5.8.
- ❻ `__call__` também é proveniente do exemplo 5.8. Ele não é necessário para satisfazer a interface de `Tombola`, mas adicionar métodos extras não faz mal algum.

BingoCage herda o método custoso `loaded` e o `inspect` tolo de `Tombola`. Ambos poderiam ser sobrescritos com funções rápidas de uma linha, como no exemplo 11.13. A questão é: podemos ser preguiçosos e simplesmente herdar os métodos concretos, possivelmente ineficientes, de uma ABC. Os métodos herdados de `Tombola` não são tão rápidos quanto poderiam ser para `BingoCage`, mas oferecem resultados corretos para qualquer subclasse de `Tombola` que implemente `pick` e `load` corretamente.

O exemplo 11.13 mostra uma implementação muito diferente, embora seja igualmente válida, da interface de `Tombola`. Em vez de embaralhar as “bolas” e extrair a última, `LotteryBlower` faz a extração de uma posição aleatória.

Exemplo 11.13 – `lotto.py`: `LotteryBlower` é uma subclasse concreta que sobrescreve os métodos `inspect` e `loaded` de `Tombola`

```
import random

from tombola import Tombola


class LotteryBlower(Tombola):
    def __init__(self, iterable):
        self._balls = list(iterable) ❶

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
            position = random.randrange(len(self._balls)) ❷
        except ValueError:
            raise LookupError('pick from empty BingoCage')
        return self._balls.pop(position) ❸

    def loaded(self): ❹
        return bool(self._balls)

    def inspect(self): ❺
        return tuple(sorted(self._balls))
```

- ❶ O inicializador aceita qualquer iterável: o argumento é usado para criar uma lista.
- ❷ A função `random.randrange(...)` gera `ValueError` se o intervalo estiver vazio, portanto nós o capturamos e lançamos `LookupError` em seu lugar para ser compatível com `Tombola`.
- ❸ Caso contrário, o item selecionado aleatoriamente é extraído de `self._balls`.
- ❹ Sobrescreve `loaded` para evitar a chamada a `inspect` (como faz `Tombola.loaded` no exemplo 11.9). Podemos deixá-lo mais rápido trabalhando com `self._balls` diretamente – não é preciso construir todo um `tuple` ordenado.
- ❺ Sobrescreve `inspect` com uma só linha.

O exemplo 11.13 mostra uma prática que já vimos: em `__init__`, `self._balls` armazena `list(iterable)`, e não apenas uma referência a `iterable` (ou seja, não atribuímos simplesmente `iterable` a `self._balls`). Como mencionamos antes¹³, isso torna nosso

¹³ Apresentei isso como exemplo de duck typing depois da seção “Aves aquáticas e ABCs”, de Martelli, na página 359.

`LotteryBlower` flexível porque o argumento `iterable` pode ser qualquer tipo iterável. Ao mesmo tempo, garantimos que armazenaremos seus itens em uma `list` para que possamos fazer `pop` dos itens. Mesmo que sempre recebêssemos listas como o argumento `iterable`, `list(iterable)` gera uma cópia do argumento, o que é uma boa prática, considerando que removeremos itens dela e talvez o cliente não espere que a `list` de itens que ele forneceu seja alterada.¹⁴

Chegamos agora à característica dinâmica crucial de `goose typing`: declarar subclasses virtuais com o método `register`.

Uma subclasse virtual de `Tombola`

Uma característica essencial de `goose typing` – e o motivo pelo qual essa técnica merece um nome de ave aquática – é a possibilidade de registrar uma classe como uma *subclasse virtual* de uma ABC mesmo que ela não herde dessa classe. Ao fazer isso, prometemos que a classe implementa fielmente a interface definida na ABC – e Python acreditará em nós sem conferir. Se mentirmos, seremos pegos pelas exceções normais em tempo de execução.

Isso é feito chamando um método `register` da ABC. A classe registrada torna-se então uma subclasse virtual da ABC e será reconhecida dessa maneira por funções como `issubclass` e `isinstance`, mas não herdará nenhum método ou atributo da ABC.



As subclasses virtuais não herdam de suas ABCs registradas, e em nenhum momento Python verifica se elas estão em conformidade com a interface da ABC, nem mesmo quando elas são instanciadas. Cabe à subclasse implementar todos os métodos necessários para evitar erros em tempo de execução.

O método `register` normalmente é chamado como uma função simples (veja a seção “Uso de `register` na prática” na página 384), mas ele também pode ser usado como um decorador. No exemplo 11.14, usamos a sintaxe de decorador e implementamos `TomboList`, que é uma subclasse virtual de `Tombola`, representada na figura 11.5.

`TomboList` funciona conforme anunciado, e os doctests que provam isso estão descritos na seção “Como as subclasses de `Tombola` foram testadas” na página 381.

¹⁴ A seção “Programação defensiva com parâmetros mutáveis” na página 272 do capítulo 8 foi dedicada ao problema de apelidos (*aliasing*) que acabamos de evitar aqui.

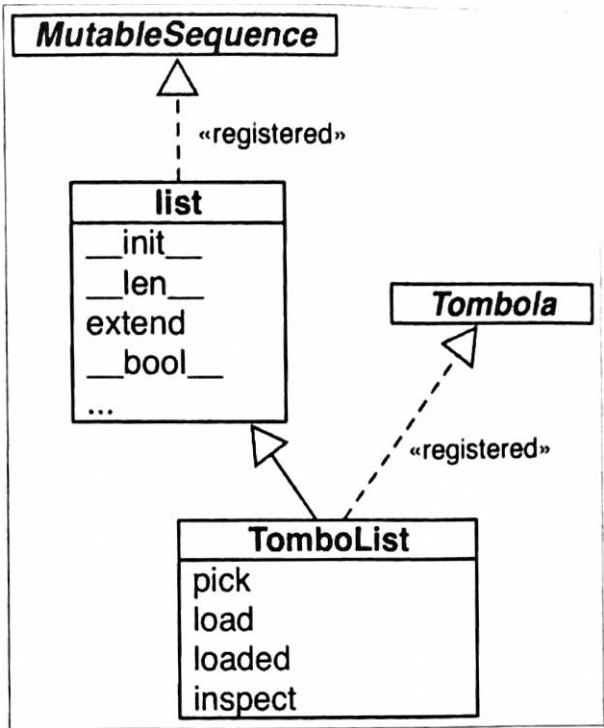


Figura 11.5 – Diagrama de classes UML para TomboList, que é uma subclasse real de list e uma subclasse virtual de Tombola.

Exemplo 11.14 – tombolist.py: a classe TomboList é uma subclasse virtual de Tombola

```
from random import randrange
from tombola import Tombola

@Tombola.register # ❶
class TomboList(list): # ❷
    def pick(self):
        if self: # ❸
            position = randrange(len(self))
            return self.pop(position) # ❹
        else:
            raise LookupError('pop from empty TomboList')
    load = list.extend # ❺
    def loaded(self):
        return bool(self) # ❻
    def inspect(self):
        return tuple(sorted(self))

# Tombola.register(TomboList) # ❼
```

- ❶ Tombolist é registrada como uma subclasse virtual de Tombola.
- ❷ Tombolist estende list.

- ❸ `Tombolist` herda `__bool__` de `list`, e esse método devolve `True` se a lista não estiver vazia.
- ❹ Nosso `pick` chama `self.pop`, herdado de `list`, passando um índice aleatório de item.
- ❺ `Tombolist.load` é igual a `list.extend`.
- ❻ `loaded` delega para `bool`.¹⁵
- ❼ Se você usa Python 3.3 ou uma versão mais antiga, não poderá usar `.register` como decorador de classe. Terá que usar a sintaxe-padrão de chamada de método.

Observe que, por causa do registro, as funções `issubclass` e `isinstance` agem como se `TomboList` fosse uma subclasse de `Tombola`:

```
>>> from tombola import Tombola
>>> from tombolist import TomboList
>>> issubclass(TomboList, Tombola)
True
>>> t = TomboList(range(100))
>>> isinstance(t, Tombola)
True
```

Entretanto a herança é orientada por um atributo de classe especial chamado `__mro__` – o Method Resolution Order (Ordem de Resolução de Métodos). Basicamente, ele lista a classe e suas superclasses na ordem usada por Python para procurar os métodos.¹⁶ Se você inspecionar o `__mro__` de `TomboList`, verá que ele lista somente as superclasses “reais” – `list` e `object`:

```
>>> TomboList.__mro__
(<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)
```

`Tombola` não está em `TomboList.__mro__`, portanto `TomboList` não herda nenhum método de `Tombola`.

Como escrevi classes diferentes para implementar a mesma interface, gostaria de ter uma maneira de submetê-las ao mesmo conjunto de doctests. A próxima seção mostra como aproveitei a API de classes normais e de ABCs para isso.

¹⁵ O mesmo truque que usei com `load` não funciona com `loaded` porque o tipo `list` não implementa `__bool__`, o método que eu deveria associar a `loaded`. Por outro lado, a função embutida `bool` não precisa de `__bool__` para funcionar porque ela também pode usar `__len__`. Veja a seção “4.1. Truth Value Testing” (Testes de valor de verdade, <https://docs.python.org/3/library/stdtypes.html#truth>) no capítulo “Built-in Types” (Tipos embutidos).

¹⁶ Há toda uma seção que explica o atributo de classe `__mro__` em “Herança múltipla e ordem de resolução de métodos” na página 399. Neste momento, essa explicação rápida deverá ser suficiente.

Como as subclasses de Tombola foram testadas

O script que usei para testar os exemplos de Tombola utiliza dois atributos de classe que permitem a introspecção de uma hierarquia de classes:

`__subclasses__()`

Método que devolve uma lista das subclasses imediatas da classe. A lista não inclui as subclasses virtuais.

`_abc_registry`

Atributo de dados – disponível somente em ABCs – associado a um `WeakSet` com referências fracas a subclasses virtuais registradas na classe abstrata.

Para testar todas as subclasses de Tombola, escrevi um script para fazer uma iteração por uma lista criada a partir de `Tombola.__subclasses__()` e de `Tombola._abc_registry`, e associei cada classe ao nome `ConcreteTombola` usado nos doctests.

Uma execução bem-sucedida do script de teste gera esta saída:

```
$ python3 tombola_runner.py
BingoCage      23 tests,  0 failed - OK
LotteryBlower  23 tests,  0 failed - OK
TumblingDrum   23 tests,  0 failed - OK
TomboList      23 tests,  0 failed - OK
```

O script de teste está no exemplo 11.15 e os doctests estão no exemplo 11.16.

Exemplo 11.15 – `tombola_runner.py`: executor de testes para as subclasses de Tombola

```
import doctest

from tombola import Tombola

# módulos para testar
import bingo, lotto, tombolist, drum ❶

TEST_FILE = 'tombola_tests.rst'
TEST_MSG = '{0:16} {1.attempted:2} tests, {1.failed:2} failed - {2}'

def main(argv):
    verbose = '-v' in argv
    real_subclasses = Tombola.__subclasses__() ❷
    virtual_subclasses = list(Tombola._abc_registry) ❸
    for cls in real_subclasses + virtual_subclasses: ❹
        test(cls, verbose)
```

```
def test(cls, verbose=False):
    res = doctest.testfile(
        TEST_FILE,
        globs={'ConcreteTombola': cls}, ❸
        verbose=verbose,
        optionflags=doctest.REPORT_ONLY_FIRST_FAILURE)
    tag = 'FAIL' if res.failed else 'OK'
    print(TEST_MSG.format(cls.__name__, res, tag)) ❹

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

- ❶ Importa os módulos contendo subclasses reais ou virtuais de Tombola para testes.
- ❷ `__subclasses__()` lista os descendentes diretos vivos na memória. É por isso que importamos os módulos para testar, mesmo que não haja outras menções a eles no código-fonte: para carregar as classes na memória.
- ❸ Cria uma list a partir de `_abc_registry` (que é um `WeakSet`) para que possamos concatená-la ao resultado de `__subclasses__()`.
- ❹ Faz a iteração pelas subclasses encontradas, passando cada item à função `test`.
- ❺ O argumento `cls` – a classe a ser testada – está associada ao nome `ConcreteTombola` no namespace global fornecido para executar o `doctest`.
- ❻ O resultado do teste é exibido com o nome da classe, o número dos testes executados, os testes que falharam e um rótulo 'OK' ou 'FAIL'.

O arquivo de doctests está no exemplo 11.16.

Exemplo 11.16 – `tombola_tests.rst`: doctests para as subclasses de Tombola

```
=====
```

Testes de Tombola

```
=====
```

Toda subclasse concreta de Tombola deve passar nesses testes

Cria e carrega instância a partir de um iterável::

```
>>> balls = list(range(3))
>>> globe = ConcreteTombola(balls)
>>> globe.loaded()
True
>>> globe.inspect()
(0, 1, 2)
```

Escolhe e reúne as bolas::

```
>>> picks = []
>>> picks.append(globe.pick())
>>> picks.append(globe.pick())
>>> picks.append(globe.pick())
```

Verifica estado e resultados::

```
>>> globe.loaded()
False
>>> sorted(picks) == balls
True
```

Recarrega::

```
>>> globe.load(balls)
>>> globe.loaded()
True
>>> picks = [globe.pick() for i in balls]
>>> globe.loaded()
False
```

Verifica se `LookupError` (ou uma subclasse) é a exceção lançada quando o dispositivo está vazio::

```
>>> globe = ConcreteTombola([])
>>> try:
...     globe.pick()
... except LookupError as exc:
...     print('OK')
OK
```

Carrega e seleciona cem bolas para verificar se todas foram escolhidas::

```
>>> balls = list(range(100))
>>> globe = ConcreteTombola(balls)
>>> picks = []
>>> while globe.inspect():
...     picks.append(globe.pick())
>>> len(picks) == len(balls)
True
>>> set(picks) == set(balls)
True
```


Verifica se a ordem mudou, e não está simplesmente invertida::

```
>>> picks != balls
True
>>> picks[::-1] != balls
True
```

Observação: os dois testes anteriores têm uma chance **muito** pequena de falhar, mesmo que a implementação esteja correta. A probabilidade de cem bolas serem escolhidas por acaso na ordem em que foram inspecionadas é de 1/100!, ou aproximadamente 1.07e-158. É muito mais fácil ganhar na mega-sena ou ficar bilionário trabalhando como programador.

FIM

Com isso, concluímos nosso estudo de caso da ABC Tombola. Na próxima seção, falaremos do uso que se faz da função de ABC *register* no mundo real.

Uso de *register* na prática

No exemplo 11.14, usamos `Tombola.register` como um decorador de classe. Antes de Python 3.3, *register* não podia ser usado dessa maneira – ele tinha que ser chamado como uma função simples após a definição da classe, conforme sugerido pelo comentário no final do exemplo 11.14.

Entretanto, mesmo que *register* atualmente possa ser chamado como um decorador, ele é mais amplamente usado como uma função para registrar classes definidas em outros locais. Por exemplo, no código-fonte (<http://bit.ly/1QOA3Lt>) do módulo `collections.abc`, os tipos embutidos `tuple`, `str`, `range` e `memoryview` são registrados como subclasses virtuais de `Sequence`, assim:

```
Sequence.register(tuple)
Sequence.register(str)
Sequence.register(range)
Sequence.register(memoryview)
```

Vários outros tipos embutidos estão registrados em ABCs em `_collections_abc.py` (<http://bit.ly/1QOA3Lt>). Esses registros ocorrem somente quando esse módulo é importado, o que não é um problema, porque você terá que importá-lo, de qualquer maneira, para obter as ABCs; é preciso acessar `MutableMapping` para poder escrever `isinstance(my_dict, MutableMapping)`.

Encerraremos este capítulo explicando um pouco da magia com ABCs que Alex Martelli usou em “Aves aquáticas e ABCs” na página 360.

Gansos podem se comportar como patos

Em seu texto *Aves aquáticas e ABCs*, Alex mostrou que uma classe pode ser reconhecida como uma subclasse virtual de uma ABC mesmo sem registro. Eis o seu exemplo novamente, com um teste adicional usando `issubclass`:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abc.Sized)
True
```

A classe `Struggle` é considerada uma subclasse de `abc.Sized` pela função `issubclass` (e, consequentemente, por `isinstance` também), pois `abc.Sized` implementa um método especial de classe chamado `__subclasshook__`. Veja o exemplo 11.17.

Exemplo 11.17 – Definição de `Sized` do código-fonte de `Lib/_collections_abc.py` (Python 3.4, <http://bit.ly/1QOG4aP>)

```
class Sized(metaclass=ABCMeta):
    __slots__ = ()
    @abstractmethod
    def __len__(self):
        return 0
    @classmethod
    def __subclasshook__(cls, C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__): # ❶
                return True # ❷
            return NotImplemented # ❸
```

- ❶ Se houver um atributo chamado `__len__` no `__dict__` de qualquer classe listada em `C.__mro__` (isto é, em `C` e em suas superclasses)...
- ❷ ... devolve `True`, indicando que `C` é uma subclasse virtual de `Sized`.
- ❸ Caso contrário, devolve `NotImplemented` para deixar a verificação de subclasses continuar.

Se você estiver interessado nos detalhes da verificação de subclasse, veja o código-fonte do método `ABCMeta.__subclasscheck__` em `Lib/abc.py` (<https://hg.python.org/cpython/file/3.4/Lib/abc.py#l194>). Já vou avisando: ele tem muitos ifs e duas chamadas recursivas.

`__subclasshook__` mistura um pouco do DNA de duck typing no esquema de goose typing. Você pode ter definições formais de interface com ABCs, pode fazer verificação com `isinstance` em qualquer lugar e continua tendo uma classe totalmente não relacionada cooperando somente porque ela implementa determinado método (ou porque ela faz o que for necessário para convencer um `__subclasshook__` a atestar que ela é uma subclasse virtual). É claro que isso só funciona para ABCs que ofereçam um `__subclasshook__`.

É uma boa ideia implementar `__subclasshook__` em nossas próprias ABCs? Provavelmente não. Todas as implementações de `__subclasshook__` que já vi no código-fonte de Python estão em ABCs como `Sized`, que declaram apenas um método especial, e elas verificam somente esse nome de método especial. Dado o status de “especial”, você pode estar bem certo de que qualquer método chamado `__len__` faz o que se espera dele. Mas, mesmo nos domínios dos métodos especiais e das ABCs fundamentais, fazer esse tipo de suposição pode ser arriscado. Por exemplo, os mapeamentos implementam `__len__`, `__getitem__` e `__iter__`, mas certamente não são considerados um subtipo de `Sequence`, porque você não pode recuperar itens usando um `offset` numérico inteiro, e eles não oferecem garantias quanto à ordem dos itens – exceto, é claro, no caso de `OrderedDict`, que preserva o ordem de inserção, mas também aceita a recuperação de itens por `offset`.

Para ABCs que você e eu venhamos a escrever, um `__subclasshook__` seria menos confiável ainda. Não estou pronto para confiar que qualquer classe chamada `Spam` que implemente ou herde `load`, `pick`, `inspect` e `loaded` vai se comportar como uma `Tombola`. É melhor deixar o programador afirmar isso criando uma subclasse `Spam` de `Tombola` ou, no mínimo, registrando-a: `Tombola.register(Spam)`. É claro que seu `__subclasshook__` também poderia verificar as assinaturas dos métodos e outros indícios, mas simplesmente não acho que vale a pena.

Resumo do capítulo

O objetivo deste capítulo foi dar um passeio a partir da natureza altamente dinâmica das interfaces informais – chamadas de protocolos – visitando as declarações estáticas de interface das ABCs e concluir com o lado dinâmico das ABCs: subclasses virtuais e detecção dinâmica de subclasses com `__subclasshook__`.

Iniciamos a jornada analisando o entendimento tradicional de interfaces na comunidade Python. Na maior parte da história de Python, tivemos consciência de interfaces, mas elas eram informais como os protocolos de `Smalltalk`, e a documentação oficial usava termos como “protocolo de foo”, “interface de foo” e “objeto foo ou similar”

indistintamente. As interfaces em estilo de protocolo não têm nenhuma relação com herança; cada classe é independente na implementação de um protocolo. É assim que são as interfaces quando adotamos duck typing.

No exemplo 11.3, observamos com que profundidade Python trata o protocolo de sequências. Se uma classe implementa `__getitem__` e nada mais, Python é capaz de fazer uma iteração por ela, e o operador `in` simplesmente funciona. Em seguida, voltamos ao velho exemplo de `FrenchDeck` do capítulo 1 para implementar a operação de embaralhar as cartas adicionando um método dinamicamente. Isso mostrou o monkey patching e enfatizou a natureza dinâmica dos protocolos. Novamente, vimos como um protocolo implementado parcialmente pode ser útil: apenas adicionar `__setitem__` do protocolo de sequência mutável nos permitiu aproveitar uma função da biblioteca-padrão, pronta para uso: `random.shuffle`. Conhecer os protocolos existentes nos permite aproveitar o máximo da rica biblioteca-padrão de Python.

Em seguida, Alex Martelli apresentou o termo “goose typing”¹⁷ para descrever um novo estilo de programação em Python. Com “goose typing”, as ABCs são usadas para deixar as interfaces explícitas, e as classes podem afirmar que implementam uma interface por serem subclasses de uma ABC ou registrando-se junto a elas – sem a necessidade da ligação forte e estática de um relacionamento de herança.

O exemplo com `FrenchDeck2` deixou claras as principais vantagens e desvantagens de ABCs explícitas. Herdar de `abc.MutableSequence` nos forçou a implementar dois métodos de que não precisávamos realmente: `insert` e `__delitem__`. Por outro lado, mesmo um novato em Python pode olhar para `FrenchDeck2` e ver que ela é uma sequência mutável. Além disso, como bônus, herdamos 11 métodos prontos para uso de `abc.MutableSequence` (cinco indiretamente de `abc.Sequence`).

Após uma visão panorâmica das ABCs existentes em `collections.abc` na figura 11.3, criamos uma ABC do zero. Doug Hellmann, criador do interessante `PyMOTW.com` (Python Module of the Week, ou Módulo Python da semana em <http://pymotw.com/>), explica a motivação:

Ao definir uma classe-base abstrata, uma API comum pode ser definida para um conjunto de subclasses. Essa capacidade é especialmente útil em situações em que alguém com menos familiaridade com o código-fonte de uma aplicação vá oferecer extensões plug-in...¹⁸

Colocando a ABC `Tombola` em uso, criamos três subclasses concretas: duas que herdam de `Tombola` e outra como uma subclasse virtual registrada junto a ela; todas elas passaram pelo mesmo conjunto de testes.

¹⁷ Alex cunhou a expressão “goose typing”, e esta é a primeira vez que ela aparece em um livro!

¹⁸ Página do módulo `abc` em `PyMOTW`, seção “Why use Abstract Base Classes?” (Por que usar classes-base abstratas, <http://bit.ly/1QOGle5>).

Ao concluir o capítulo, mencionamos como diversos tipos embutidos estão registrados em ABCs no módulo `collections.abc`; desse modo, você pode usar `isinstance(memoryview, abc.Sequence)` e obter `True`, mesmo que `memoryview` não herde de `abc.Sequence`. Por fim, vimos a magia de `__subclasshook__`, que permite a uma ABC reconhecer qualquer classe não registrada como uma subclasse, desde que ela passe em um teste que pode ser tão simples ou complexo quanto se queira – os exemplos da biblioteca-padrão simplesmente verificam os nomes dos métodos.

Para acrescentar, gostaria de reforçar o conselho de Alex Martelli, de evitar a criação de nossas próprias ABCs, exceto quando estivermos criando frameworks extensíveis pelo usuário – o que, na maior parte do tempo, não é nosso trabalho. No uso cotidiano, nosso contato com as ABCs deverá se restringir a criar subclasses ou registrar classes junto a ABCs existentes. Com menos frequência que criar subclasses ou registrar, podemos usar ABCs para verificações com `isinstance`. Raramente – se é que algum dia o faremos – teremos a oportunidade de criar uma nova ABC do zero.

Depois de 15 anos usando Python, a primeira classe abstrata que escrevi, que não era apenas um exemplo didático, foi a classe `Board` (<https://github.com/garoa/pingo/blob/master/pingo/board.py>) do projeto Pingo (<http://pingo.io/>). Os drivers que dão suporte a diferentes computadores de uma só placa e controladores são subclasses de `Board`, compartilhando, assim, a mesma interface. Na verdade, embora tenha sido concebida e implementada como uma classe abstrata, a classe `pingo.Board` não é subclasse de `abc.ABC` atualmente (quando escrevi este livro).¹⁹ Pretendo fazer de `Board` uma ABC explícita em algum momento – mas há tarefas mais importantes a fazer no projeto.

Eis uma citação apropriada para finalizar este capítulo:

Embora as ABCs facilitem a verificação de tipos, não é algo que deva ser usado de modo exagerado em um programa. Em sua essência, Python é uma linguagem dinâmica que oferece uma enorme flexibilidade. Tentar impor restrições de tipo em todos os lugares tende a resultar em códigos mais complicados que o necessário. Você deve abraçar a flexibilidade de Python.²⁰

— David Beazley e Brian Jones
Python Cookbook

Ou como o revisor técnico Leonardo Rochael escreveu: “Se você se sentir tentado a criar sua própria ABC, por favor, tente primeiro resolver seu problema por meio de duck typing normal.”

¹⁹ Você verá isso na biblioteca-padrão de Python também: classes que, na verdade, são abstratas, mas ninguém as criou explicitamente dessa maneira.

²⁰ *Python Cookbook*, 3ª edição (O'Reilly), “Recipe 8.12. Defining an Interface or Abstract Base Class” (Receita 8.12. Definir uma interface ou uma classe base abstrata), p. 276.

Leituras complementares

O livro *Python Cookbook*, 3ª ed. (O'Reilly, <http://shop.oreilly.com/product/0636920027072.do>)²¹ de Beazley e Jones tem uma seção sobre definição de ABC (Receita 8.12). O livro foi escrito antes de Python 3.4, portanto não usa a sintaxe atual preferível para declarar ABCs, que é criar subclasses de `abc.ABC`, em vez de usar a palavra reservada `metaclass`. Exceto por esse pequeno detalhe, a receita inclui a maior parte dos recursos de ABC, descrevendo-as muito bem, e termina com o valioso conselho citado no final da seção anterior.

The Python Standard Library by Example de Doug Hellmann (Addison-Wesley) tem um capítulo sobre o módulo `abc`. Também está disponível na Web no excelente *PyMOTW – Python Module of the Week* (Módulo Python da semana, <http://pymotw.com/2/abc/index.html>) de Doug. Tanto o livro quanto o site focam em Python 2; desse modo, ajustes devem ser feitos se você estiver usando Python 3. Em Python 3.4, lembre-se de que o único decorador de método recomendado para ABC é `@abstractmethod` – os demais se tornaram obsoletos. A outra citação sobre ABCs no resumo do capítulo é do site e do livro de Doug.

Ao usar ABCs, herança múltipla não só é comum como também é praticamente inevitável, pois cada uma das ABCs fundamentais de coleção – `Sequence`, `Mapping` e `Set` estende várias ABCs (veja a figura 11.3). Desse modo, o capítulo 12 é uma continuação importante para este capítulo.

A PEP 3119 — *Introducing Abstract Base Classes* (Introdução às classes-base abstratas, <https://www.python.org/dev/peps/pep-3119>) oferece a justificativa por trás das ABCs, e a PEP 3141 — *A Type Hierarchy for Numbers* (Uma hierarquia de tipos para números, <https://www.python.org/dev/peps/pep-3141>) apresenta as ABCs do módulo `numbers` (<https://docs.python.org/3/library/numbers.html>).

Para uma discussão sobre os prós e contras da tipagem dinâmica, veja a entrevista de Guido van Rossum a Bill Venners em “Contracts in Python: A Conversation with Guido van Rossum, Part IV” (Contratos em Python: uma conversa com Guido van Rossum, Parte IV, <http://www.artima.com/intv/pycontract.html>).

O pacote `zope.interface` (<http://docs.zope.org/zope.interface/>) oferece uma maneira de declarar interfaces, verificando se os objetos as implementam, registrando provedores e consultando provedores de uma dada interface. O pacote começou como uma parte central do framework Zope 3, mas pode e tem sido usado fora do Zope. É a base da arquitetura flexível de componentes de projetos Python de larga escala como Twisted, Pyramid e Plone. Lennart Regebro escreveu uma ótima introdução ao `zope.interface` em “A Python Component Architecture” (A arquitetura de um componente Python,

²¹ N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

<http://bit.ly/1QOH6x>). Baiju M escreveu um livro todo sobre o assunto: *A Comprehensive Guide to Zope Component Architecture* (<http://muthukadan.net/docs/zca.html>).

Ponto de vista

Type hints ou sinalização de tipos

Provavelmente, a maior novidade no mundo Python em 2014 foi o sinal verde dado por Guido van Rossum à implementação de verificação opcional de tipagem estática usando anotações de função, de modo semelhante ao que o verificador Mypy (<http://www.mypy-lang.org/>) faz. Isso aconteceu na lista de discussão Python-ideas em 15 de agosto. A mensagem é *Optional static typing — the crossroads* (Tipagem estática opcional – a encruzilhada, <http://bit.ly/1QOHhyX>). No mês seguinte, a PEP 484 - *Type Hints* (Sinalizações de tipos, <https://www.python.org/dev/peps/pep-0484/>), cujo autor é Guido, foi publicada como versão preliminar.

A ideia é permitir que programadores usem opcionalmente anotações para declarar parâmetros e tipos de retorno em definições de função. A palavra-chave, nesse caso, é *opcionalmente*. Você só usará essas anotações se quiser os benefícios e as restrições que as acompanham, e poderá colocá-las em algumas funções e não em outras, a seu critério.

Superficialmente, isso pode soar como o que a Microsoft fez com TypeScript, seu superconjunto de JavaScript, exceto que TypeScript vai mais longe: ela adiciona novas construções à linguagem (por exemplo, módulos, classes, interfaces explícitas etc.), permite declarações de variáveis com tipo e, na verdade, é compilado para JavaScript. Quando escrevi este livro, os objetivos da tipagem estática opcional em Python eram muito menos ambiciosos.

Para entender o alcance dessa proposta, há um ponto-chave enfatizado por Guido no email histórico de 15 de agosto de 2014:

Farei uma suposição adicional: os principais casos de uso serão linting, IDEs e geração de doc. Todos eles têm um ponto em comum: deve ser possível executar um programa, mesmo que a verificação de tipos falhe. Além disso, adicionar tipos a um programa não deverá atrapalhar o seu desempenho (nem contribuirá com ele :-).

Portanto parece que esse não é um passo tão radical quanto parecia ser à primeira vista. A PEP 482 – *Literature Overview for Type Hints* (Visão geral da literatura para sinalizações de tipos, <https://www.python.org/dev/peps/pep-0482/>) é referenciada pela PEP 484 – *Type Hints* (Sinalizações de tipos, <https://www.python.org/dev/peps/pep-0484/>) e documenta rapidamente as sinalizações de tipos em ferramentas Python de terceiros e em outras linguagens.

Radical ou não, as sinalizações de tipos estão chegando: o suporte à PEP 484 na forma de um módulo `typing` será incluído já em Python 3.5. A forma como a proposta está redigida e será implementada deixa claro que códigos existentes não deixarão de executar por falta de sinalizações de tipos – ou pela sua adição. Por fim, a PEP 484 afirma claramente que:

Deve-se enfatizar também que Python continuará sendo uma linguagem dinamicamente tipada, e os autores não desejam, de modo algum, tornar as sinalizações de tipo obrigatórias, mesmo por convenção.

Python tem tipagem fraca?

Discussões sobre as regras de tipagem em linguagens às vezes são confusas por falta de uma terminologia uniforme. Alguns escritores (como Bill Venners na entrevista com Guido mencionada na seção “Leituras complementares” na página 389) dizem que Python tem tipagem fraca, o que a coloca na mesma categoria de JavaScript e PHP. Uma maneira melhor de falar sobre as regras de tipagem é considerar dois eixos diferentes:

Tipagem forte versus fraca

Se a linguagem raramente faz conversão implícita de tipos, ela é considerada uma linguagem de tipagem forte: se faz isso com frequência, é de tipagem fraca. Java, C++ e Python têm tipagem forte. PHP, JavaScript e Perl têm tipagem fraca.

Tipagem estática versus dinâmica

Se a verificação de tipos é realizada em tempo de compilação, a linguagem tem tipagem estática; se isso ocorrer em tempo de execução, tem tipagem dinâmica. A tipagem estática exige declarações de tipo (algumas linguagens modernas usam inferência de tipo para evitar parte das declarações). Fortran e Lisp são as duas linguagens de programação mais antigas ainda vivas, e elas usam tipagem estática e dinâmica, respectivamente.

A tipagem forte ajuda a identificar bugs com antecedência.

Eis alguns exemplos que mostram por que a tipagem fraca é ruim:²²

```
// isto é JavaScript (testado com Node.js v0.10.33)
'' == '0'    // falso
0 == ''      // verdadeiro
0 == '0'     // verdadeiro
'' < 0       // falso
'' < '0'     // verdadeiro
```

²² Adaptado do livro *JavaScript: The Good Parts* (O'Reilly) de Douglas Crockford, Apêndice B, p. 109.

Python não faz coerção automática entre strings e números, portanto as expressões `==` resultam em `False` – preservando a transitividade de `==` – e as comparações com `<` levantam `TypeError` em Python 3.

A tipagem estática facilita às ferramentas (compiladores, IDEs) analisar códigos para detectar erros e oferecer outros serviços (otimização, refatoração etc.). A tipagem dinâmica aumenta as oportunidades para reutilização de código, reduz o número de linhas e permite que as interfaces surjam naturalmente como protocolos, em vez de serem impostas previamente.

Para resumir, Python usa tipagem dinâmica e forte. A *PEP 484 – Type Hints* (Sinalizações de tipos, <https://www.python.org/dev/peps/pep-0484/>) não mudará isso, mas permitirá aos autores de APIs adicionar anotações opcionais de tipo para que as ferramentas possam realizar algumas verificações estáticas de tipos.

Monkey patching

Monkey patching tem uma péssima reputação. Se houver abuso, seu uso pode resultar em sistemas difíceis de entender e de manter. O patch normalmente está altamente acoplado ao seu alvo, tornando-o rígido e quebradiço. Outro problema é que duas bibliotecas que apliquem monkey-patches podem interferir uma na outra, com a segunda biblioteca executada destruindo os patches da primeira.

Porém o monkey patching também pode ser útil, por exemplo, para fazer uma classe implementar um protocolo em tempo de execução. O padrão de projeto Adapter (Adaptador) resolve o mesmo problema implementando uma classe totalmente nova.

É fácil usar monkey-patch em código Python, mas há limitações. Ao contrário de Ruby e de JavaScript, Python não permite fazer monkey-patch em tipos embutidos. Na verdade, considero isso uma vantagem, pois você pode ter certeza de que um objeto `str` sempre terá os mesmos métodos com o mesmo comportamento. Essa limitação diminui as chances de bibliotecas externas tentarem aplicar patches conflitantes.

Interfaces em Java, Go e Ruby

Desde 1989, as classes abstratas têm sido usadas para especificar interfaces na linguagem C++. Os designers de Java optaram por não ter herança múltipla de classes, o que impossibilitou o uso de classes abstratas como especificações de interface – porque, geralmente, uma classe precisa implementar mais de uma

interface. Porém eles adicionaram `interface` como construção da linguagem, e uma classe pode implementar mais de uma interface – uma forma de herança múltipla. Deixar as definições de interface mais explícitas do que nunca foi uma excelente contribuição de Java. Com Java 8, uma interface pode oferecer implementações de métodos, chamadas de Default Methods (Métodos default, <https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html>). Com isso, as interfaces Java se tornaram mais parecidas com as classes abstratas de C++ e de Python.

A linguagem Go usa uma abordagem totalmente diferente. Em primeiro lugar, não há herança em Go. Você pode definir interfaces, mas não precisa (e, na verdade, não pode) dizer explicitamente que um determinado tipo implementa uma interface. O compilador determina isso automaticamente. Então o que há em Go pode ser chamado de “duck typing estático”, no sentido de que as interfaces são verificadas em tempo de compilação, mas o que importa é o que os tipos de fato implementam.

Comparando com Python, é como se, em Go, toda ABC implementasse o método `__subclasshook__`, verificando os nomes e as assinaturas dos métodos implementados nos tipos, e você jamais criasse subclasses ou registrasse uma ABC. Se quiséssemos que Python se parecesse mais com Go, teríamos de fazer verificação de tipos nos argumentos de todas as funções. Parte da infraestrutura está disponível (lembre-se da seção “Anotações de função” na página 192). Guido já disse que usar essas anotações para verificação de tipos não seria um problema – pelo menos em ferramentas de apoio. Veja a seção “Ponto de vista” na página 202 do capítulo 5 para saber mais.

Os Rubyistas acreditam firmemente em duck typing, e Ruby não tem um modo formal de declarar uma interface ou uma classe abstrata, exceto fazendo o mesmo que fazíamos em Python antes da versão 2.6: levantar `NotImplementedError` no corpo dos métodos para torná-los abstratos, forçando o usuário a criar subclasses e implementá-las.

Nesse meio-tempo, li que Yukihiro “Matz” Matsumoto, criador de Ruby, disse em uma palestra em setembro de 2014 que a tipagem estática poderá estar no futuro da linguagem. Isso aconteceu no Ruby Kaigi no Japão, uma das conferências anuais mais importantes de Ruby. Ainda não vi uma transcrição, mas Godfrey Chan fez um post sobre isso em seu blog: “Ruby Kaigi 2014: Day 2” (<http://brewhouse.io/blog/2014/09/19/ruby-kaigi-2014-day-2>). A partir do relato de Chan, parece que Matz focou em anotações de função. Há até mesmo menções às anotações de função em Python.

Fico me perguntando se as anotações de função seriam realmente úteis sem as ABCs para acrescentar estrutura ao sistema de tipos sem perder flexibilidade. Sendo assim, as interfaces formais talvez estejam também no futuro de Ruby.

Acredito que as ABCs de Python, com a função `register` e `__subclasshook__`, trouxeram as interfaces formais à linguagem sem abrir mão das vantagens da tipagem dinâmica.

Talvez os gansos estejam destinados a tomar o lugar dos patos.

Metáforas e “idioms” em interfaces

Uma metáfora promove o entendimento ao deixar claras as restrições. Esse é o valor das palavras “pilha” e “fila” para descrever essas estruturas fundamentais de dados: elas deixam claro como os itens podem ser adicionados e removidos. Por outro lado, Alan Cooper escreveu no livro *About Face, 4E* (Wiley):

A aderência rígida às metáforas amarra desnecessariamente as interfaces ao funcionamento do mundo físico.

Ele está se referindo às interfaces de usuário, mas a advertência aplica-se também às APIs. Entretanto Cooper concorda que, quando uma metáfora “realmente apropriada cai em nosso colo”, podemos usá-la (ele escreve “cai em nosso colo” porque é difícil encontrar metáforas adequadas e você não deve gastar tempo esforçando-se para procurá-las). Acredito que a imagem do sistema de bingo que usei neste capítulo seja apropriada, e defendo o seu uso.

About Face, de longe, é o melhor livro que já li sobre design de UI – e olhe que já li alguns. A coisa mais importante que aprendi com a obra de Cooper foi abrir mão das metáforas como paradigma de design e substituí-las por “interfaces idiomáticas”. Conforme mencionado, Cooper não lida com APIs, mas quanto mais penso em suas ideias, mais vejo como elas se aplicam em Python. Os protocolos fundamentais da linguagem são o que Cooper chama de “idioms”. Depois que aprendemos o que é uma “sequência”, podemos aplicar esse conhecimento em diferentes contextos. Esse é o tema principal de *Python fluente*: destacar os “idioms” fundamentais da linguagem para que seu código seja conciso, eficiente e legível – para um pythonista fluente.

CAPÍTULO 12

Herança: para o bem ou para o mal

Começamos a insistir na ideia de herança como uma maneira de permitir que novatos desenvolvessem com base em frameworks que poderiam ser projetados somente por experts.¹

— Alan Kay

The Early History of Smalltalk

Este capítulo é sobre herança e subclasses com ênfase em duas particularidades muito específicas de Python:

- Armadilhas na criação de subclasses a partir de tipos embutidos
- Herança múltipla e a ordem de resolução de métodos

Muitos consideram que a herança múltipla representa mais problemas que vantagens. Sua ausência certamente não atrapalhou o sucesso de Java; é provável que tenha ajudado na adoção generalizada dessa linguagem, depois de muitos terem ficado traumatizados pelo uso exagerado de herança múltipla em C++.

Entretanto, por causa do incrível sucesso e da influência de Java, muitos programadores chegam até Python sem ter visto herança múltipla na prática. É por isso que, em vez de usar exemplos apenas didáticos, nossa discussão sobre herança múltipla será ilustrada por dois projetos Python importantes: o kit de desenvolvimento para GUIs Tkinter e o web framework Django.

Começaremos pela questão de criar subclasses de tipos embutidos. O restante do capítulo abordará a herança múltipla com nossos estudos de caso e discutirá práticas boas e ruins na criação de hierarquias de classes.

¹ Alan Kay, “The Early History of Smalltalk” (História do início de Smalltalk) em SIGPLAN Not. 28, 3 (março de 1993), 69–95. Também disponível online (<http://propella.sakura.ne.jp/earlyHistoryST/EarlyHistoryST.html>). Agradeço ao meu amigo Christiano Anderson, que compartilhou essa referência quando escrevi este capítulo.

Artimanhas da criação de subclasses de tipos embutidos

Antes de Python 2.2, não era possível criar subclasses de tipos embutidos como `list` ou `dict`. Desde então, isso pode ser feito, mas há uma ressalva importante: o código dos tipos embutidos (escrito em C) não chama métodos especiais sobrescritos pelas classes definidas pelo usuário.

Uma boa descrição rápida do problema está na documentação de PyPy em “Differences between PyPy and CPython” (Diferenças entre PyPy e CPython) na seção *Subclasses of built-in types* (Subclasses de tipos embutidos, <http://bit.ly/1JHNmhX>):

Oficialmente, CPython não tem nenhuma regra para quando, exatamente, métodos sobrescritos em subclasses de tipos embutidos são implicitamente chamados ou não. Como regra aproximada, esses métodos jamais são chamados por outros métodos embutidos do mesmo objeto. Por exemplo, um `__getitem__()` sobrescrito em uma subclasse de `dict` não será chamado pelo método embutido `get()`.

O exemplo 12.1 mostra o problema.

Exemplo 12.1 – Nossa versão sobrescrita de `__setitem__` é ignorada pelos métodos `__init__` e `__update__` do tipo embutido `dict`

```
>>> class DoppelDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2) # ❶
...
>>> dd = DoppelDict(one=1) # ❷
>>> dd
{'one': 1}
>>> dd['two'] = 2 # ❸
>>> dd
{'one': 1, 'two': [2, 2]}
>>> dd.update(three=3) # ❹
>>> dd
{'three': 3, 'one': 1, 'two': [2, 2]}
```

- ❶ `DoppelDict.__setitem__` duplica valores no armazenamento (sem um bom motivo, somente para ter um efeito visível). Ele funciona delegando para a superclasse.
- ❷ O método `__init__` herdado de `dict` claramente ignorou que `__setitem__` foi sobrescrito: o valor de `'one'` não é duplicado.
- ❸ O operador `[]` chama nosso `__setitem__` e funciona conforme esperado: `'two'` é mapeado para o valor duplicado `[2, 2]`.
- ❹ O método `update` de `dict` também não usa nossa versão de `__setitem__`: o valor de `'three'` não foi duplicado.

O comportamento desse tipo embutido viola uma regra básica da programação orientada a objetos: a busca de métodos sempre deve começar na classe da instância-alvo (`self`), mesmo quando a chamada ocorre dentro de um método implementado em uma superclasse. Nessa triste situação, o método `__missing__` – que vimos na seção “Método `__missing__`” na página 102 – funciona conforme documentado somente porque é tratado como um caso especial.

O problema não está limitado a chamadas em uma instância – se `self.get()` chama `self.__getitem__()` –, mas ocorre também com métodos sobrescritos de outras classes que deveriam ser chamados pelos métodos dos tipos embutidos. O exemplo 12.2 mostra um exemplo adaptado da documentação de PyPy (<http://bit.ly/1JHNmhX>).

Exemplo 12.2 – `__getitem__` de `AnswerDict` é ignorado por `dict.update`

```
>>> class AnswerDict(dict):
...     def __getitem__(self, key): # ❶
...         return 42
...
>>> ad = AnswerDict(a='foo') # ❷
>>> ad['a'] # ❸
42
>>> d = {}
>>> d.update(ad) # ❹
>>> d['a'] # ❺
'foo'
>>> d
{'a': 'foo'}
```

- ❶ `AnswerDict.__getitem__` sempre devolve 42, independentemente da chave.
- ❷ `ad` é um `AnswerDict` carregado com o par chave-valor ('a', 'foo').
- ❸ `ad['a']` devolve 42, como esperado.
- ❹ `d` é uma instância vazia de `dict`, que atualizamos com `ad`.
- ❺ O método `dict.update` ignorou nosso `AnswerDict.__getitem__`.



Criar subclasses de tipos embutidos como `dict` ou `list` ou `str` diretamente é propenso a erros porque os métodos de tipos embutidos geralmente ignoram as versões sobrescritas definidas pelo usuário. Em vez de criar subclasses de tipos embutidos, derive suas classes das classes `UserDict`, `UserList` e `UserString` do módulo `collections` (<http://docs.python.org/3/library/collections.html>). Elas existem para serem facilmente estendidas.

Se você criar uma subclasse de `collections.UserDict` em vez de `dict`, os problemas expostos nos exemplos 12.1 e 12.2 serão ambos corrigidos. Veja o exemplo 12.3.

Exemplo 12.3 – `DoppelDict2` e `AnswerDict2` funcionam conforme esperado porque estendem `UserDict`, e não `dict`

```
>>> import collections
>>>
>>> class DoppelDict2(collections.UserDict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)
...
>>> dd = DoppelDict2(one=1)
>>> dd
{'one': [1, 1]}
>>> dd['two'] = 2
>>> dd
{'two': [2, 2], 'one': [1, 1]}
>>> dd.update(three=3)
>>> dd
{'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
>>>
>>> class AnswerDict2(collections.UserDict):
...     def __getitem__(self, key):
...         return 42
...
>>> ad = AnswerDict2(a='foo')
>>> ad['a']
42
>>> d = {}
>>> d.update(ad)
>>> d['a']
42
>>> d
{'a': 42}
```

Como experimento para medir o trabalho extra exigido na criação de uma subclasse de um tipo embutido, reescrevi a classe `StrKeyDict` do exemplo 3.8. A versão original herdava de `collections.UserDict` e implementava somente três métodos: `__missing__`, `__contains__` e `__setitem__`. A classe experimental `StrKeydict` derivada de `dict` implementa os mesmos três métodos, com pequenos ajustes, por causa da maneira como os dados são armazenados. Mas para que ela passasse pela mesma suíte de testes, tive

de implementar `__init__`, `get` e `update` porque as versões herdadas de `dict` se recusavam a cooperar com os métodos `__missing__`, `__contains__` e `__setitem__` sobrescritos. A subclasse de `UserDict` do exemplo 3.8 tem 16 linhas, enquanto a subclasse experimental de `dict` ficou com 37 linhas.²

Para resumir: o problema descrito nesta seção aplica-se apenas à delegação de métodos na implementação em linguagem C dos tipos embutidos e afeta somente as classes definidas pelo usuário diretamente derivadas desses tipos. Se criar uma subclasse de uma classe escrita em Python, por exemplo, `UserDict` ou `MutableMapping`, você não terá problemas.³

Outra questão relacionada à herança, particularmente à herança múltipla, é: como Python decide qual atributo será usado se superclasses em ramos paralelos definirem atributos com o mesmo nome? A resposta será dada a seguir.

Herança múltipla e ordem de resolução de métodos

Qualquer linguagem que implemente herança múltipla deve lidar com potenciais conflitos de nomes se classes ancestrais não relacionadas entre si implementarem um método de mesmo nome. Isso é chamado de “problema do losango” (diamond problem), conforme mostram a figura 12.1 e o exemplo 12.4.

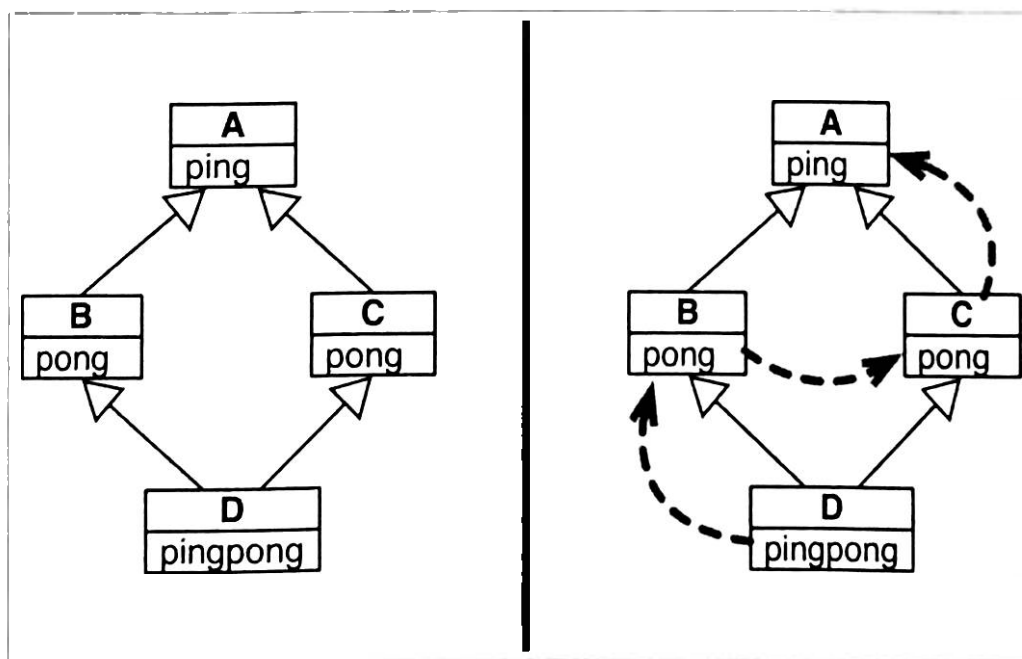


Figura 12.1 – À esquerda: diagrama UML de classes mostrando o “problema do losango”. À direita: as setas tracejadas representam a MRO (Method Resolution Order, ou Ordem de Resolução de Métodos) de Python para o exemplo 12.4.

- ² Se estiver curioso, o experimento está no arquivo `strkeydict_dicts.py` no repositório de código de Python fluente (<https://github.com/fluentpython/example-code>).
- ³ A propósito, quanto a essa questão, PyPy comporta-se de modo mais “correto” que CPython ao custo de introduzir uma pequena incompatibilidade. Acesse “Differences between PyPy and CPython” (Diferenças entre PyPy e CPython, <http://bit.ly/1JHNmhX>) para ver os detalhes.

Exemplo 12.4 – diamond.py: as classes A, B, C e D formam o grafo da figura 12.1

```
class A:
    def ping(self):
        print('ping:', self)

class B(A):
    def pong(self):
        print('pong:', self)

class C(A):
    def pong(self):
        print('PONG:', self)

class D(B, C):
    def ping(self):
        super().ping()
        print('post-ping:', self)
    def pingpong(self):
        self.ping()
        super().ping()
        self.pong()
        super().pong()
        C.pong(self)
```

Observe que tanto a classe B quanto a classe C implementam um método pong. A única diferença é que C.pong exibe a palavra PONG em letras maiúsculas.

Se você chamar d.pong() em uma instância de D, qual método pong será executado? Em C++, é obrigatório qualificar as chamadas de métodos colocando explicitamente nomes de classe para resolver essa ambiguidade. Isso também pode ser feito em Python. Dê uma olhada no exemplo 12.5.

Exemplo 12.5 – Duas maneiras de chamar o método pong em uma instância da classe D

```
>>> from diamond import *
>>> d = D()
>>> d.pong() # ❶
pong: <diamond.D object at 0x10066c278>
>>> C.pong(d) # ❷
PONG: <diamond.D object at 0x10066c278>
```

❶ Chamar simplesmente d.pong() faz a versão de B ser executada.

- ❶ Você sempre pode chamar um método de uma superclasse diretamente, passando a instância como um argumento explícito.

A ambiguidade de uma chamada como `d.pong()` é resolvida porque Python segue uma ordem específica ao percorrer o grafo de herança. Essa ordem se chama MRO: Method Resolution Order (Ordem de Resolução de Métodos). As classes têm um atributo chamado `__mro__` com uma tupla de referências às superclasses na ordem MRO, da classe atual até a classe `object`. Para a classe `D`, este é o `__mro__` (veja a figura 12.1):

```
>>> D.__mro__
(<class 'diamond.D'>, <class 'diamond.B'>, <class 'diamond.C'>,
 <class 'diamond.A'>, <class 'object'>)
```

A maneira recomendada de delegar chamadas de métodos a superclasses é usar a função embutida `super()`, mais fácil de usar em Python 3, como mostra o método `pingpong` da classe `D` no exemplo 12.4.⁴ Mas também é possível, e às vezes conveniente, ignorar a MRO e chamar diretamente um método de uma superclasse. Por exemplo, o método `D.ping` poderia ser escrito como:

```
def ping(self):
    A.ping(self) # em vez de super().ping()
    print('post-ping:', self)
```

Observe que, ao chamar um método de instância diretamente em uma classe, você deve passar `self` explicitamente, pois está acessando um *método sem associação* (unbound method).

Entretanto, usar `super()` é mais seguro e mais resistente a mudanças futuras, especialmente quando chamamos métodos em um framework ou em qualquer hierarquia de classes sobre a qual você não tenha controle. O exemplo 12.6 mostra que `super()` segue a MRO ao chamar um método.

Exemplo 12.6 – Usando `super()` para chamar `ping` (código-fonte no exemplo 12.4)

```
>>> from diamond import D
>>> d = D()
>>> d.ping() # ❶
ping: <diamond.D object at 0x10cc40630> # ❷
post-ping: <diamond.D object at 0x10cc40630> # ❸
```

- ❶ O `ping` de `D` faz duas chamadas.
- ❷ A primeira chamada é `super().ping()`; `super` delega a chamada de `ping` à classe `A`; `A.ping` exibe essa linha.
- ❸ A segunda chamada é `print('post-ping:', self)`, que exibe essa linha.

⁴ Em Python 2, a primeira linha de `D.pingpong` seria escrita como `super(D, self).ping()` em vez de `super().ping()`.

Vamos ver agora o que acontece quando `pingpong` é chamado em uma instância de `D`. Veja o exemplo 12.7.

Exemplo 12.7 – As cinco chamadas feitas por `pingpong` (código-fonte no exemplo 12.4)

```
>>> from diamond import D
>>> d = D()
>>> d.pingpong()
>>> d.pingpong()
ping: <diamond.D object at 0x10bf235c0> # ❶
post-ping: <diamond.D object at 0x10bf235c0>
ping: <diamond.D object at 0x10bf235c0> # ❷
pong: <diamond.D object at 0x10bf235c0> # ❸
pong: <diamond.D object at 0x10bf235c0> # ❹
PONG: <diamond.D object at 0x10bf235c0> # ❺
```

- ❶ A primeira chamada é `self.ping()`; ela executa o método `ping` de `D`, que exibe essa linha e a próxima.
- ❷ A segunda chamada é `super().ping()`, que ignora o `ping` de `D` e encontra o método `ping` em `A`.
- ❸ A terceira chamada é `self.pong()`, que encontra a implementação de `pong` em `B` de acordo com a `__mro__`.
- ❹ A quarta chamada é `super().pong()`, que encontra a mesma implementação de `B.pong`, também de acordo com a `__mro__`.
- ❺ A quinta chamada é `C.pong(self)`, que encontra a implementação de `C.pong`, ignorando a `__mro__`.

A MRO leva em conta não só o grafo de herança, mas também a ordem em que as superclasses estão listadas na declaração da subclasse. Em outras palavras, se em *diamond.py* (Exemplo 12.4) a classe `D` fosse declarada como `class D(C, B):`, a `__mro__` da classe `D` seria diferente; `C` seria pesquisada antes de `B`.

Muitas vezes, verifico a `__mro__` das classes interativamente quando as estou estudando. O exemplo 12.8 tem alguns exemplos com classes conhecidas.

Exemplo 12.8 – Inspeccionando o atributo `__mro__` de várias classes

```
>>> bool.__mro__ ❶
(<class 'bool'>, <class 'int'>, <class 'object'>)
>>> def print_mro(cls): ❷
...     print(', '.join(c.__name__ for c in cls.__mro__))
... 
```

```

>>> print_mro(bool)
bool, int, object
>>> from frenchdeck2 import FrenchDeck2
>>> print_mro(FrenchDeck2) ❸
FrenchDeck2, MutableSequence, Sequence, Sized, Iterable, Container, object
>>> import numbers
>>> print_mro(numbers.Integral) ❹
Integral, Rational, Real, Complex, Number, object
>>> import io ❺
>>> print_mro(io.BytesIO)
BytesIO, _BufferedIOBase, _IOBase, object
>>> print_mro(io.TextIOWrapper)
TextIOWrapper, _TextIOBase, _IOBase, object

```

- ❶ `bool` herda métodos e atributos de `int` e de `object`.
- ❷ `print_mro` gera exibições mais compactas da MRO.
- ❸ Os ancestrais de `FrenchDeck2` incluem várias ABCs do módulo `collections.abc`.
- ❹ Essas são as ABCs numéricas incluídas no módulo `numbers`.
- ❺ O módulo `io` inclui ABCs (aquelas com sufixo `...Base`) e classes concretas como `BytesIO` e `TextIOWrapper`, que são os tipos para objetos-arquivos binário e texto devolvidos por `open()`, de acordo com o argumento de modo.



A MRO é calculada com um algoritmo chamado C3. O artigo canônico sobre a MRO de Python que explica o C3 é “The Python 2.3 Method Resolution Order” (A ordem de resolução de métodos em Python 2.3, <http://bit.ly/1OwVqBd>) de Michele Simionato. Se estiver interessado nos detalhes da MRO, a seção “Leituras complementares” na página 415 tem outras referências. Mas não se preocupe muito com isso, pois o algoritmo é sensato. Como escreve Simionato:

[...] a menos que faça uso intenso de herança múltipla e tenha hierarquias não triviais, você não precisará entender o algoritmo C3 e poderá simplesmente ignorar este artigo.

Para encerrar a discussão sobre MRO, a figura 12.2 mostra parte do complicado grafo de herança múltipla do pacote para GUI Tkinter da biblioteca-padrão de Python. Para estudar a figura, comece pela classe `Text` na parte inferior. A classe `Text` implementa um campo de texto editável completo para múltiplas linhas. O campo tem muitas funcionalidades próprias, mas também herda muitos métodos de outras classes. O lado esquerdo mostra um diagrama de classes UML resumido. À direita, ele está decorado com setas que mostram a MRO, conforme listado a seguir com a ajuda da função auxiliar `print_mro` definida no exemplo 12.8:

```
>>> import tkinter
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

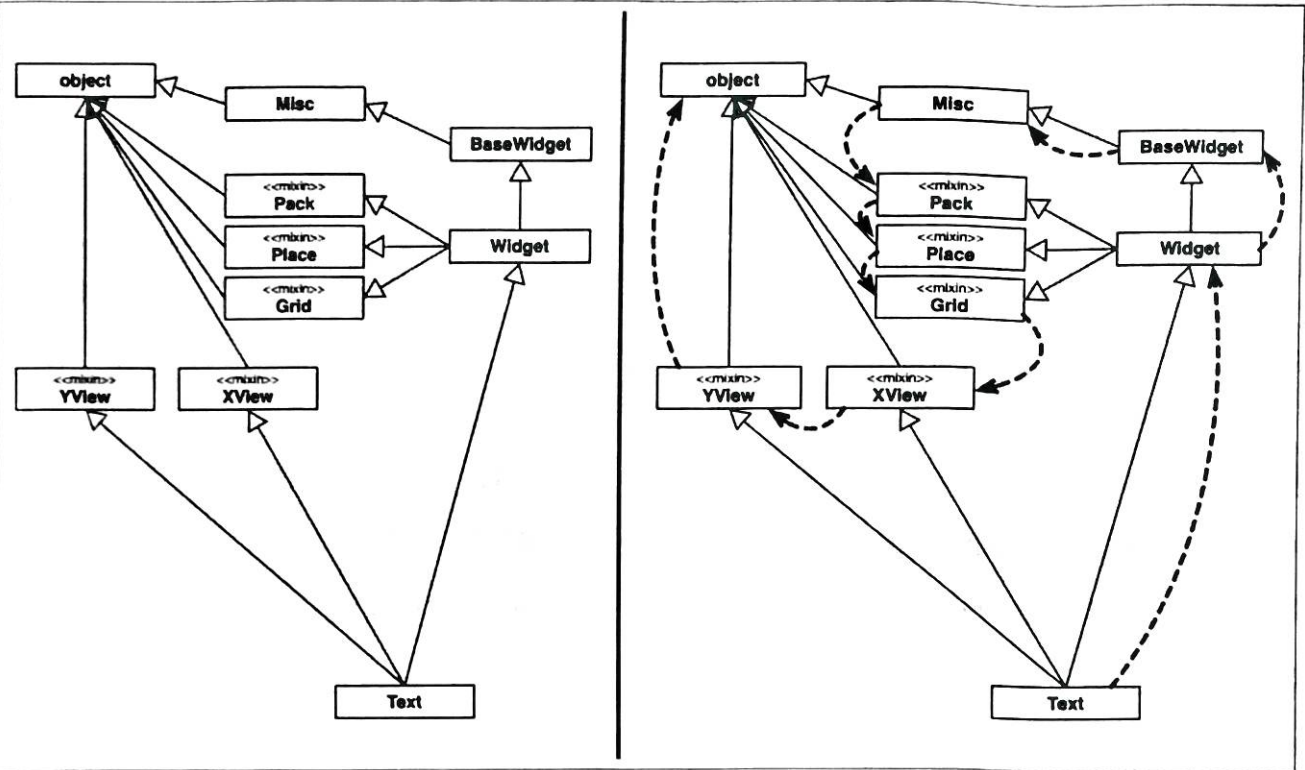


Figura 12.2 – À esquerda: diagrama de classes UML mostrando a classe de widget Text de Tkinter e suas superclasses. À direita: as setas tracejadas representam Text.__mro__.

Na próxima seção, discutiremos os prós e contras da herança múltipla, com exemplos de frameworks reais que a usam.

Herança múltipla no mundo real

É possível fazer bom uso da herança múltipla. O padrão Adapter (Adaptador) do livro *Design Patterns*⁵ usa herança múltipla, então fazer isso não deve estar totalmente errado (os outros 22 padrões do livro usam apenas herança simples, portanto, claramente a herança múltipla não é a cura para todos os males).

Na biblioteca-padrão de Python, o uso mais visível de herança múltipla está no pacote `collections.abc`. Não é um fato controverso: afinal de contas, até mesmo Java tem suporte para herança múltipla de interfaces, e as ABCs são declarações de interfaces que, opcionalmente, podem oferecer implementações de métodos concretos.⁶

5 N.T.: Edição brasileira publicada com o título “Padrões de projeto” (Bookman).
6 Conforme mencionamos anteriormente, Java 8 permite que as interfaces também ofereçam implementações de métodos. O novo recurso chama-se Default Methods (<http://bit.ly/1JHPsyk>) no Tutorial de Java oficial.

Um exemplo extremo de herança múltipla na biblioteca-padrão é o Tkinter [módulo `tkinter`: *Python interface to Tcl/Tk* (Interface Python para Tcl/Tk, <https://docs.python.org/3/library/tkinter.html>)]. Usei parte da hierarquia de classes de Tkinter para mostrar a MRO da figura 12.2, mas a figura 12.3 exhibe todas as classes de widgets (controles) do pacote básico `tkinter` (há mais widgets no subpacote `tkinter.ttk`, <https://docs.python.org/3/library/tkinter.ttk.html>).

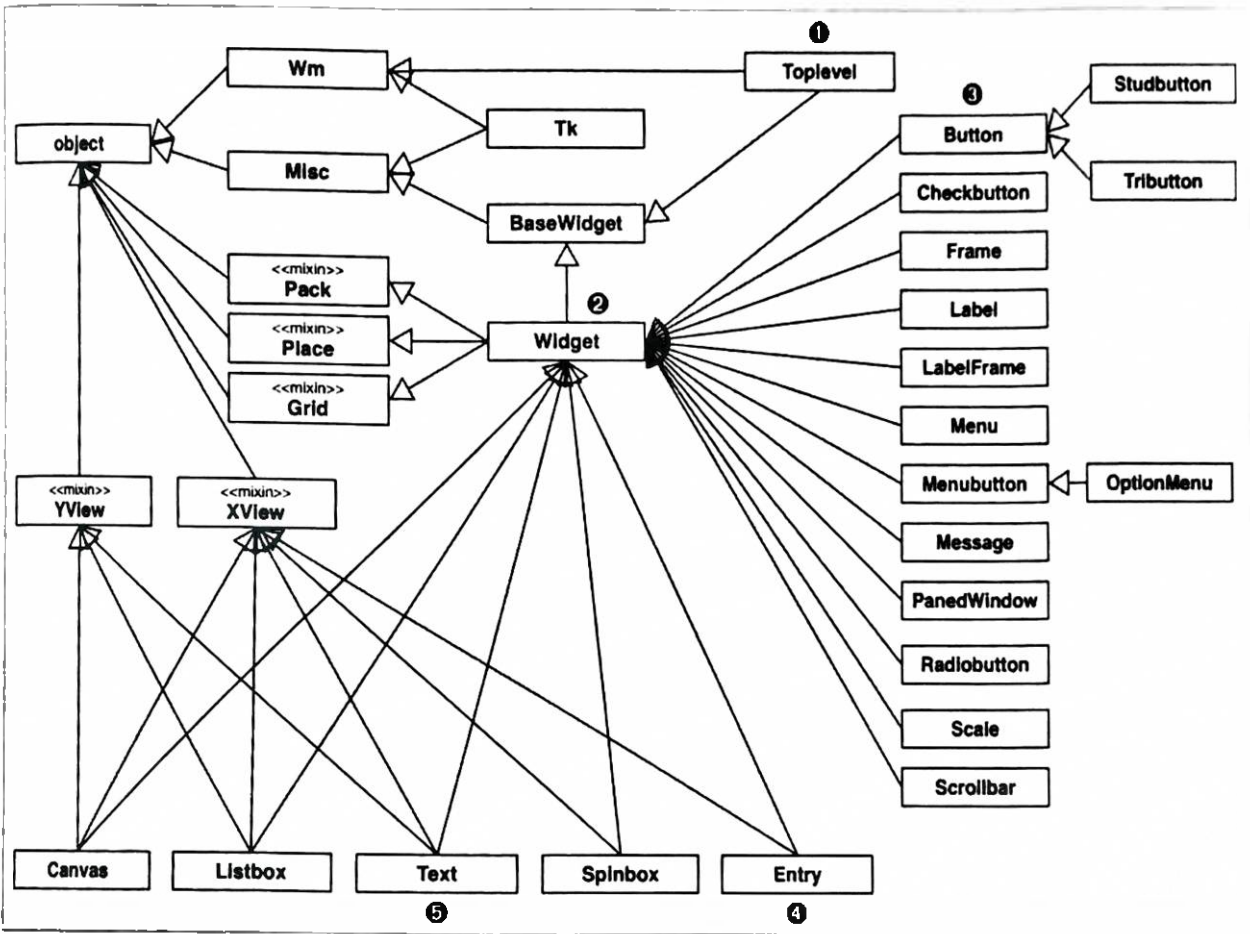


Figura 12.3 – Diagrama UML resumido da hierarquia de classes da ferramenta GUI Tkinter; as classes com a etiqueta «mixin» foram projetadas para oferecer métodos concretos a outras classes por meio de herança múltipla.

Tkinter tem vinte anos de idade atualmente (quando escrevi este livro), e não é um exemplo das melhores práticas hoje em dia. Contudo ele mostra como a herança múltipla era usada quando os programadores não consideravam suas desvantagens. E servirá como contraexemplo quando discutirmos algumas boas práticas na próxima seção.

Considere as seguintes classes da figura 12.3:

- ❶ Toplevel: a classe de uma janela de mais alto nível em uma aplicação Tkinter.
- ❷ Widget: a superclasse de todos os objetos visíveis que podem ser colocados em uma janela.
- ❸ Button: o widget de um botão simples.

- ❹ `Entry`: um campo de texto editável de uma só linha.
- ❺ `Text`: um campo de texto editável para múltiplas linhas.

Veja as MROs dessas classes, exibidas pela função `print_mro` do exemplo 12.8:

```
>>> import tkinter
>>> print_mro(tkinter.Toplevel)
Toplevel, BaseWidget, Misc, Wm, object
>>> print_mro(tkinter.Widget)
Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Button)
Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Entry)
Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

Veja alguns aspectos que podem ser observados sobre o relacionamento dessas classes com as demais:

- `Toplevel` é a única classe gráfica que não herda de `Widget` porque é a janela de nível mais alto e não se comporta como um widget – por exemplo, ela não pode ser posicionada dentro de outra janela ou frame. `Toplevel` herda de `Wm`, que oferece funções para acesso direto ao gerenciador de janelas do host, por exemplo, para definir o título da janela e configurar suas bordas.
- `Widget` herda diretamente de `BaseWidget` e de `Pack`, `Place` e `Grid`. Essas três últimas classes são gerenciadores de geometria: são responsáveis pela disposição dos widgets em uma janela ou frame. Cada uma encapsula uma estratégia de layout e uma API diferente para posicionamento de widgets.
- `Button`, como a maioria dos widgets, é descendente apenas de `Widget`, mas indiretamente de `Misc` – que proporciona dezenas de métodos a todos os widgets.
- `Entry` é subclasse de `Widget` e de `XView`, a classe que implementa rolagem horizontal.
- `Text` é subclasse de `Widget`, `XView` e `YView`, que oferece a funcionalidade de rolagem vertical.

Discutiremos algumas boas práticas de herança múltipla e veremos se Tkinter as adota.

Lidando com herança múltipla

[...] precisávamos de toda uma teoria melhor para herança (e continuamos precisando). Por exemplo, a herança e a instanciação (que é um tipo de herança) confundem tanto a pragmática (por exemplo, fatorar código para economizar espaço) quanto a semântica (usada para um número excessivo de tarefas como: especialização, generalização, especiação etc.).

— Alan Kay
The Early History of Smalltalk

Como escreveu Alan Kay, a herança é usada por motivos diferentes e a herança múltipla acrescenta alternativas e complexidade. É fácil criar designs incompreensíveis e frágeis usando herança múltipla. Na falta de uma teoria abrangente, apresentamos a seguir algumas dicas para evitar grafos macarrônicos de classes.

1. Faça a distinção entre herança de interface e herança de implementação

Ao lidar com herança múltipla, é conveniente manter clara a razão pela qual estamos criando cada subclasse. Os principais motivos são:

- Herança de interface, para criar um subtipo, implicando em um relacionamento “é um” (is-a).
- Herança de implementação, para evitar duplicação de código por meio de reutilização.

Na prática, ambos os usos muitas vezes são simultâneos, mas sempre que puder deixar a intenção clara, faça isso. A herança para reutilização de código é um detalhe de implementação e frequentemente pode ser substituída por composição e delegação. Por outro lado, a herança de interface é a espinha dorsal de um framework.

2. Deixe as interfaces explícitas com ABCs

Em Python moderno, se uma classe foi projetada para definir uma interface, ela deve ser uma ABC explícita. Em Python 3.4 ou em versões mais recentes, isso quer dizer: crie uma subclasse de `abc.ABC` ou de outra ABC (veja a seção “Detalhes de sintaxe das ABCs” na página 374 se precisar oferecer suporte a versões mais antigas de Python).

3. Use mixins para reutilização de código

Se uma classe foi projetada para oferecer implementações de métodos a serem reutilizados por várias subclasses não relacionadas, sem implicar em um relacionamento “é

um”, ela deve ser uma *classe mixin* explícita. Conceitualmente, uma mixin não define um tipo novo; ela simplesmente reúne métodos para reutilização. Uma mixin jamais deve ser instanciada, e as classes concretas não devem herdar somente de uma mixin. Cada mixin deve oferecer um comportamento único específico, implementando poucos métodos com grande afinidade entre si.

4. Explícite as mixins pelo nome

Não há uma maneira formal em Python de definir que uma classe é uma mixin, portanto é altamente recomendável que elas sejam nomeadas com um sufixo `..Mixin`. Tkinter não segue esse conselho, mas, se seguisse, `XView` seria `XViewMixin`, `Pack` seria `PackMixin` e assim por diante, para todas as classes em que coloquei o rótulo «mixin» na figura 12.3.

5. Uma ABC também pode ser uma mixin; o contrário não é verdade

Como uma ABC pode implementar métodos concretos, ela funciona também como uma mixin. Uma ABC também define um tipo, o que não ocorre com uma mixin. Além disso, uma ABC pode ser a única classe-base de qualquer outra classe, enquanto uma mixin jamais deve ser a única classe-base de uma subclasse, exceto se essa subclasse for outra mixin mais especializada – uma configuração incomum em códigos de verdade.

Há uma restrição que se aplica às ABCs, mas não às mixins: os métodos concretos implementados em uma ABC devem colaborar somente com os métodos da mesma ABC e de suas superclasses. Isso implica que os métodos concretos de uma ABC são sempre para conveniência, pois tudo que eles fazem um usuário da classe também pode fazer chamando outros métodos da ABC.

6. Não herde de mais de uma classe concreta

As classes concretas não devem ter nenhuma superclasse concreta ou devem ter no máximo uma.⁷ Em outras palavras, todas as superclasses de uma classe concreta, exceto uma, devem ser ABCs ou mixins. Por exemplo, no código a seguir, se `Alpha` é uma classe concreta, então `Beta` e `Gamma` devem ser ABCs ou mixins:

```
class MyConcreteClass(Alpha, Beta, Gamma):  
    """Esta é uma classe concreta: ela pode ser instanciada."""  
    # ... mais código ...
```

⁷ No texto “Aves aquáticas e ABCs” na página 360, Alex Martelli usa uma citação de Scott Meyer, extraída de *More Effective C++*, que vai mais longe: “todas as classes que não são folhas devem ser abstratas” (isto é, classes concretas não devem ter superclasses concretas).

7. Ofereça classes agregadas aos usuários

Se alguma combinação de ABCs ou de mixins for particularmente útil ao código do cliente, ofereça uma classe que as reúna de forma sensata. Grady Booch chama isso de *classe agregada* (aggregate class).⁸

Por exemplo, veja o código-fonte completo (<http://bit.ly/IJHQqKU>) de `tkinter.Widget`:

```
class Widget(BaseWidget, Pack, Place, Grid):
    """Internal class.

    Base class for a widget which can be positioned with the
    geometry managers Pack, Place or Grid."""
    pass
```

O corpo da classe `Widget` é vazio, mas a classe oferece um serviço útil: reúne quatro superclasses para que qualquer pessoa que precise criar um novo widget não tenha de se lembrar de todos esses mixins nem fique se perguntando se eles devem ser declarados em determinada ordem em um comando `class`. Um melhor exemplo disso está na classe `ListView` de Django, que discutiremos em breve na seção “Um exemplo moderno: mixins em views genéricas de Django” na página 411.

8. “Prefira composição de objetos à herança de classe.”

Essa citação foi diretamente extraída do livro *Design Patterns*^{9 10}, e é o melhor conselho que posso oferecer aqui. Quando você se sente à vontade com herança, é muito fácil usá-la de forma abusiva. Colocar objetos em uma hierarquia organizada é atraente para o nosso senso de organização; programadores fazem isso até por mera diversão.

No entanto, preferir composição resulta em designs mais flexíveis. Por exemplo, no caso da classe `tkinter.Widget`, em vez de herdar os métodos de todos os gerenciadores de geometria, as instâncias de widgets poderiam armazenar uma referência ao gerenciador de geometria e chamar seus métodos. Afinal de contas, um `Widget` não “é” um gerenciador de geometria, mas poderia usar os serviços de um por meio de delegação. Então você poderia acrescentar um novo gerenciador de geometria sem mexer na hierarquia de classes dos widgets e sem se preocupar com conflitos de nomes. Mesmo com herança simples, esse princípio aumenta a flexibilidade, pois criar

8 “Uma classe construída principalmente por herança de mixins e que não acrescente sua própria estrutura ou comportamento é chamada de *classe agregada*.” Grady Booch et al., *Object Oriented Analysis and Design*, 3E (Addison-Wesley, 2007), p. 109.

9 Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software, Introduction*, p. 20.

10 N.T.: Edição brasileira publicada com o título “Padrões de projeto” (Bookman). A citação original é: “Favor object composition over class inheritance.”

subclasses é uma forma de acoplamento alto, e árvores altas de herança tendem a ser rígidas e quebradiças.

Composição e delegação podem substituir o uso de mixins para oferecer funcionalidades a classes diferentes, mas não podem substituir o uso de herança de interface para definir uma hierarquia de tipos.

Vamos agora analisar Tkinter do ponto de vista dessas recomendações.

Tkinter: o bom, o ruim e o feio



Tenha em mente que Tkinter faz parte da biblioteca-padrão desde o lançamento de Python 1.1 em 1994. Tkinter é uma camada sobre o excelente kit de ferramentas GUI Tk da linguagem Tcl. A combinação Tcl/Tk não é originalmente orientada a objetos, portanto a API de Tk é, basicamente, um enorme catálogo de funções. No entanto o kit de ferramentas é muito orientado a objetos em seus conceitos, mesmo não o sendo em sua implementação.

A maioria das sugestões da seção anterior não é seguida por Tkinter, com a número 7 sendo a exceção digna de nota. Mesmo assim, não é um ótimo exemplo, pois a composição provavelmente funcionaria melhor para integrar os gerenciadores de geometria em `Widget`, conforme discutido no item 8.

A docstring de `tkinter.Widget` começa com as palavras “Internal class” (Classe interna). Isso sugere que `Widget` provavelmente deveria ser uma ABC. Embora não tenha métodos próprios, `Widget` define uma interface. Sua mensagem é: “Você pode ter certeza de que todo widget de Tkinter oferece métodos básicos de widget (`__init__`, `destroy` e dezenas de funções de API de Tk), além dos métodos de todos os três gerenciadores de geometria.” Podemos concordar que essa não é uma ótima definição de interface (é ampla demais), mas é uma interface, e `Widget` a “define” como a união das interfaces de suas superclasses.

A classe `Tk`, que encapsula a lógica da aplicação GUI, herda de `Wm` e de `Misc`, e nenhuma delas é abstrata ou mixin (`Wm` não é devidamente uma mixin porque `TopLevel` é uma subclasse somente dela). O nome da classe `Misc` – por si só – é um *code smell* muito forte. `Misc` tem mais de cem métodos, e todos os widgets herdam dela. Por que é necessário que todo widget tenha métodos para tratamento de área de transferência (clipboard), seleção de texto, gerenciamento de temporizadores e funções desse tipo? Na prática, não é possível colar dados em um botão nem selecionar texto em uma barra de rolagem. `Misc` deveria estar separada em várias classes mixin especializadas, e nem todos os widgets deveriam herdar de todas essas mixins.

Para sermos justos, como usuário de Tkinter, você não precisa conhecer nem usar herança múltipla. É um detalhe de implementação oculto por trás das classes de widgets que você instanciará ou das quais criará subclasses em seu próprio código. Mas você sofrerá as consequências do uso abusivo de herança múltipla ao digitar `dir(tkinter.Button)` e tentar encontrar o método de que precisa entre os 214 atributos listados.

Apesar dos problemas, Tkinter é estável, flexível e não necessariamente feio. Os widgets legados (e defaults) de Tk não estão de acordo com as interfaces de usuário modernas, mas o pacote `tkinter.ttk` oferece widgets elegantes, de aspecto nativo, tornando viável o desenvolvimento profissional de GUI com o Tkinter desde Python 3.1 (2009). Além disso, alguns dos widgets legados, como `Canvas` e `Text`, são extremamente poderosos. Com apenas um pouco de código, podemos transformar um objeto `Canvas` em uma aplicação simples para desenhar arrastando e soltando formas. Definitivamente, vale a pena dar uma olhada em Tkinter e Tcl/Tk se você estiver interessado em programação de GUI.

No entanto nosso tema aqui não é programação de GUI, mas o uso prático de herança múltipla. Um exemplo mais atual com classes `mixin` explícitas pode ser visto em Django.

Um exemplo moderno: mixins em views genéricas de Django



Não é preciso conhecer Django para acompanhar esta seção. Estou usando apenas uma pequena parte do framework como um exemplo prático de herança múltipla e tentarei oferecer todo o conhecimento básico necessário, supondo que você tenha alguma experiência com desenvolvimento web do lado do servidor em outra linguagem ou framework.

Em Django, uma *view* é um objeto invocável (*callable*) que aceita como argumento um objeto que representa uma requisição HTTP e devolve um objeto que representa uma resposta HTTP. As diferentes respostas são o que nos interessa nessa discussão. Elas podem ser tão simples quanto uma resposta de redirecionamento, sem conteúdo no corpo, ou tão complexa quanto uma página de um catálogo em uma loja online, apresentada a partir de um *template HTML*, listando vários produtos com botões para compras e links para páginas de detalhes.

Originalmente, Django oferecia um conjunto de funções – as *views genéricas* (*generic views*) – que implementavam alguns casos de uso comuns. Por exemplo, muitos sites precisam mostrar resultados de pesquisa que incluem informações de diversos itens, com a listagem ocupando várias páginas, e para cada item, um link para uma página com informações detalhadas sobre esse item. Em Django, uma *view de lista* (*list view*) e uma *view de detalhes* (*detail view*) foram concebidas para trabalharem juntas e

resolver o seguinte problema: uma view de lista apresenta resultados de pesquisa e uma view de detalhes gera páginas para itens individuais.

Entretanto as views genéricas originais eram funções, portanto não eram extensíveis. Se fosse necessário fazer algo semelhante, mas não exatamente igual a uma view de lista genérica, seria necessário começar do zero.

Em Django 1.3, o conceito de views baseadas em classe foi introduzido, juntamente com um conjunto de classes de view genéricas, organizadas como classes-base, mixins e classes concretas prontas para uso. As classes-base e as mixins estão no módulo base do pacote `django.views.generic`, representado na figura 12.4. Na parte superior do diagrama, vemos duas classes que cuidam de responsabilidades bem distintas: `View` e `TemplateResponseMixin`.

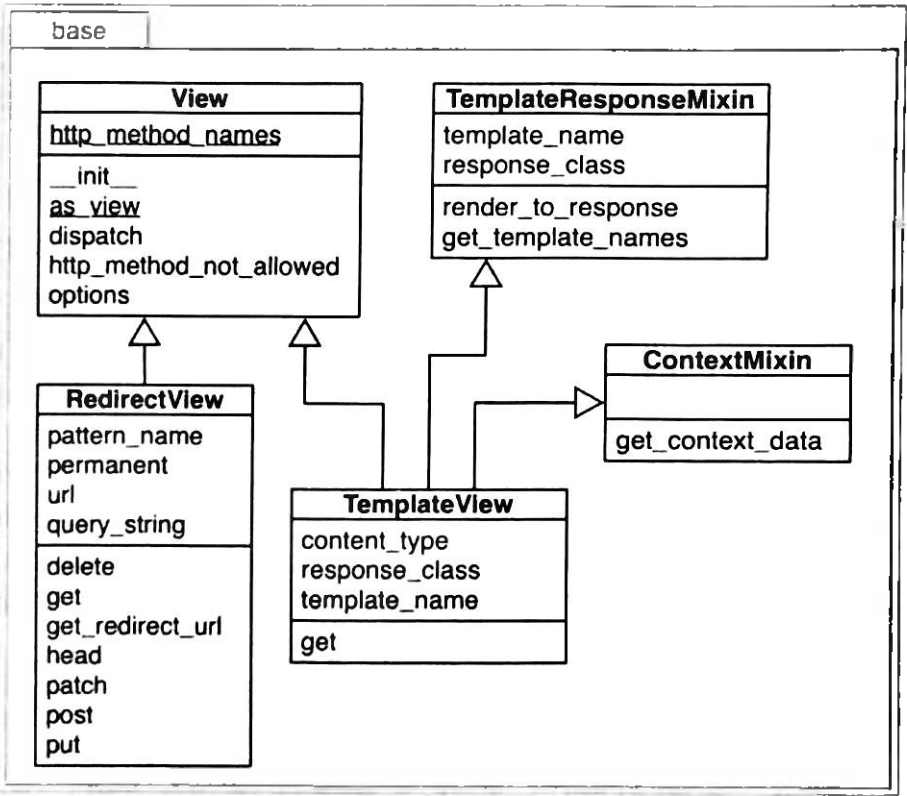


Figura 12.4 – Diagrama UML de classes do módulo `django.views.generic.base`.



Um ótimo recurso para estudar essas classes é o site *Classy Class-Based Views* (Views elegantes baseadas em classe, <http://ccbv.co.uk/>), em que você pode navegar facilmente por elas, ver todos os métodos de cada classe (métodos herdados, sobrescritos e adicionados), ver diagramas, navegar pela sua documentação e acessar o código-fonte no GitHub (<http://bit.ly/1JHSoe8>).

`View` é a classe-base de todas as views (poderia ser uma ABC) e oferece funcionalidades essenciais, como o método `dispatch`, que delega para métodos “handler” como `get`, `head`, `post` etc., implementados por subclasses concretas para tratar os diferentes

verbos HTTP.¹¹ A classe `RedirectView` herda somente de `View`, e você pode ver que ela implementa `get`, `head`, `post` etc.

Espera-se que as subclasses concretas de `View` implementem os métodos handler, então por que elas não fazem parte da interface de `View`? O motivo: as subclasses são livres para implementar apenas os handlers que quiserem tratar. Uma `TemplateView` é usada somente para exibir conteúdo, portanto ela implementa apenas `get`. Se uma requisição HTTP `post` for enviada a uma `TemplateView`, o método herdado `View.dispatch` verificará se há um handler `post` e, se não houver, gerará uma resposta `405 Method Not Allowed` (Método não permitido).¹²

`TemplateResponseMixin` oferece funcionalidades que são de interesse apenas das views que precisam usar um template. Uma `RedirectView`, por exemplo, não tem conteúdo no corpo, portanto não precisa de um template e não herda dessa mixin. `TemplateResponseMixin` oferece funcionalidades a `TemplateView` e a outras views que renderizam templates, como `ListView`, `DetailView` etc., definidas em outros módulos do pacote `django.views.generic`. A figura 12.5 representa o módulo `django.views.generic.list` e parte do módulo base.

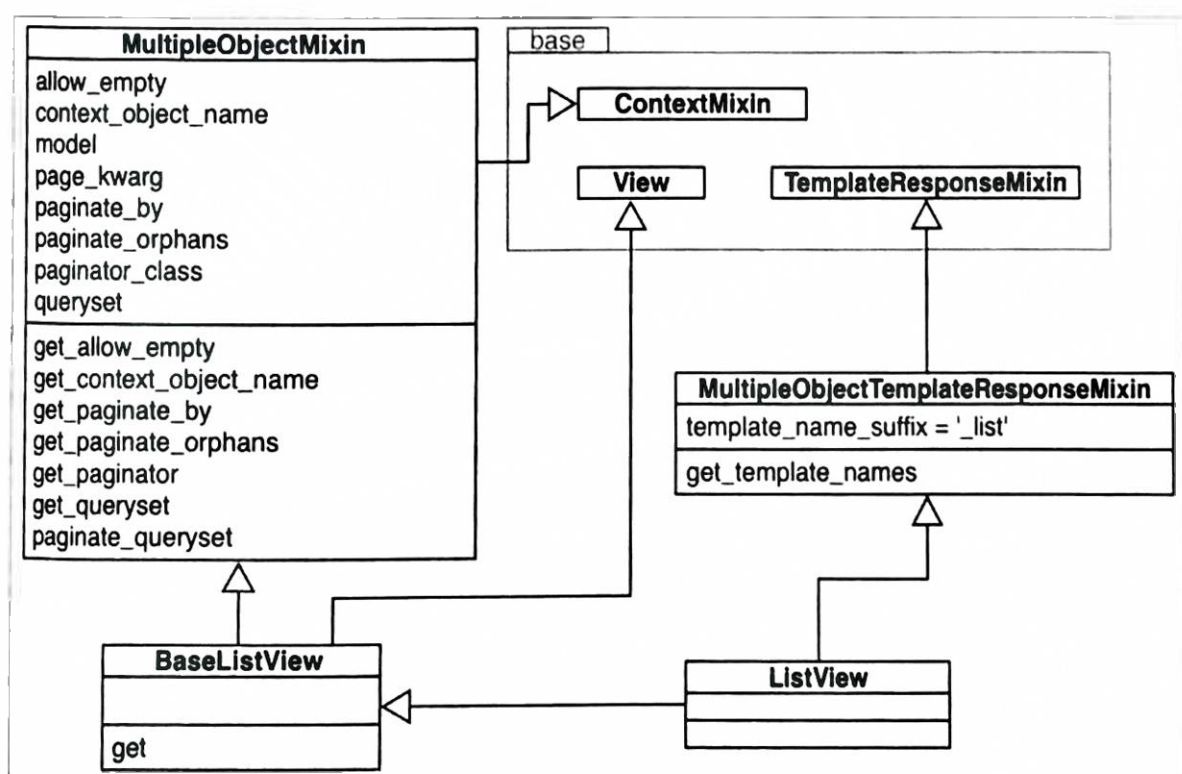


Figura 12.5 – Diagrama UML de classes do módulo `django.views.generic.list`. Nesse caso, as três classes do módulo-base não estão expandidas (veja a figura 12.4). A classe `ListView` não tem métodos nem atributos: é uma classe agregada.

¹¹ Programadores que usam Django sabem que o método de classe `as_view` é a parte mais visível da interface `View`, mas isso não é relevante para nós aqui.

¹² Se você entende de padrões de projeto, perceberá que o sistema de despacho (dispatch) de Django é uma variação dinâmica do padrão Template Method (Método Template, http://en.wikipedia.org/wiki/Template_method_pattern). É dinâmico porque a classe `View` não força as subclasses a implementar todos os handlers, mas `dispatch` verifica, em tempo de execução, se um handler concreto está disponível para a requisição específica.

Para usuários de Django, a classe mais importante na figura 12.5 é `ListView`, que é uma classe agregada, sem código (seu corpo tem apenas uma docstring). Ao ser instanciada, uma `ListView` tem um atributo de instância `object_list` com o qual o template pode fazer uma iteração e mostrar o conteúdo da página que, normalmente, é o resultado de uma consulta a um banco de dados devolvendo vários objetos. Toda a funcionalidade relacionada à geração desse iterável de objetos é proveniente de `MultipleObjectMixin`. Essa mixin também implementa a lógica complexa de paginação – exibir parte do resultado em uma página e links para mais páginas.

Suponha que você queira criar uma view que não renderizará um template, mas produzirá uma lista de objetos em formato JSON. É para isso que `BaseListView` existe. Ela oferece um ponto de extensão fácil de usar, que reúne as funcionalidades de `View` e de `MultipleObjectMixin` sem o overhead do mecanismo de template.

A API de views baseadas em classe de Django é um exemplo melhor de herança múltipla que Tkinter. Em particular, suas classes mixin são fáceis de usar em subclasses concretas: cada uma tem um propósito bem definido e são todas nomeadas com o sufixo `...Mixin`.

Views baseadas em classe não foram adotadas por todos os usuários de Django. Muitos as usam de forma limitada, como caixas-pretas, mas, quando é preciso criar algo novo, muitos programadores que usam Django continuam escrevendo funções monolíticas de view, que cuidam de todas essas responsabilidades, em vez de tentar reutilizar as views e as mixins básicas.

É preciso um pouco de tempo para aprender a aproveitar as views baseadas em classes e estendê-las de modo a atender às necessidades específicas de uma aplicação, mas acho que vale a pena estudá-las: elas eliminam muito código repetitivo, facilitando a reutilização de soluções e até mesmo melhorando a comunicação nas equipes – por exemplo, definindo nomes padronizados para os templates e para as variáveis passadas para os contextos dos templates. Views baseadas em classes são views Django “on rails”.

Com isso, concluímos nosso passeio pela herança múltipla e pelas classes mixins.

Resumo do capítulo

Começamos nossa discussão sobre herança explicando o problema com a criação de subclasses de tipos embutidos: seus métodos nativos implementados em C não chamam métodos sobrescritos em subclasses, exceto em poucos casos especiais. É por isso que, quando precisamos de um tipo `list`, `dict` ou `str` especializado, é mais fácil criar subclasses de `UserList`, `UserDict` ou `UserString` – todas definidas no módulo `collections` (<https://docs.python.org/3/library/collections.html>), que, na verdade, encapsula

os tipos embutidos e delega operações a eles –, três exemplos de favorecer a composição em vez da herança na biblioteca-padrão. Se o comportamento desejado for muito diferente do que os tipos embutidos oferecem, talvez seja mais fácil criar subclasses da ABC apropriada de `collections.abc` (<https://docs.python.org/3/library/collections.abc.html>) e fazer sua própria implementação.

O restante do capítulo foi dedicado à herança múltipla, que é uma faca de dois gumes. Inicialmente, vimos como a ordem de resolução de métodos, codificada no atributo de classe `__mro__`, trata o problema de conflitos de nomes em potencial em métodos herdados. Também vimos como a função embutida `super()` segue a `__mro__` para chamar um método de uma superclasse. Em seguida, estudamos como a herança múltipla é usada no pacote Tkinter, que faz parte da biblioteca-padrão de Python. Tkinter não é um exemplo das melhores práticas atuais, e, sendo assim, discutimos algumas maneiras de lidar melhor com herança múltipla, incluindo um uso cuidadoso de classes mixin e como evitar totalmente a herança múltipla usando composição em seu lugar. Após considerar o uso abusivo de herança múltipla em Tkinter, finalizamos estudando as partes essenciais da hierarquia de views baseadas em classe de Django, que considero ser um exemplo melhor de uso de mixins.

Lennart Regebro – um pythonista muito experiente e um dos revisores técnicos deste livro – acha confuso o design da hierarquia de views com mixin em Django. Mas ele também escreveu o seguinte:

Os perigos e as desvantagens da herança múltipla são muito exagerados. Na verdade, nunca tive nenhum grande problema com ela.

No final, cada um de nós pode ter opiniões diferentes sobre o uso de herança múltipla, ou se ela sequer deve ser usada em nossos próprios projetos. Com frequência, porém, não temos opção: os frameworks que precisamos usar impõem suas próprias escolhas.

Leituras complementares

Ao usar ABCs, a herança múltipla não só é comum como também é praticamente inevitável, pois cada uma das ABCs fundamentais de coleção (`Sequence`, `Mapping` e `Set`) estende várias ABCs. O código-fonte de `collections.abc` (`Lib/_collections_abc.py` em <http://bit.ly/1QOA3Lt>) é um bom exemplo de herança múltipla com ABCs – muitas das classes também são mixin.

O post *Python's super() considered super!* (`super()` de Python é `super!`, <http://bit.ly/1JHSZfW>) de Raymond Hettinger explica o funcionamento de `super` e da herança múltipla em Python de um ponto de vista favorável. Foi escrito em resposta a *Python's Super is nifty, but you can't use it (a.k.a. Python's Super Considered Harmful)* [`super` de Python é elegante,

mas não dá para usá-lo (também conhecido como *super* de Python considerado prejudicial, <https://fulm.net/super-harmful/>) de James Knight.

Apesar dos títulos desses posts, o problema não está realmente na função embutida *super* – que, em Python 3, não é tão feia quanto era em Python 2. O verdadeiro problema é a herança múltipla, que é inerentemente complicada e ardilosa. Michele Simionato vai além da crítica e, na realidade, oferece uma solução em seu *Setting Multiple Inheritance Straight* (Esclarecendo a herança múltipla, <http://bit.ly/1HGpYxV>): ele implementa *traits*, que é uma forma restrita de *mixins* originada na linguagem Self. Simionato tem uma longa série de posts de blog esclarecedores sobre herança múltipla em Python, incluindo *The wonders of cooperative inheritance, or using super in Python 3* (As maravilhas da herança cooperativa, ou usando *super* em Python 3, <http://bit.ly/1HGpXdj>); *Mixins considered harmful*, parte 1 (Mixins são consideradas prejudiciais, <http://bit.ly/1HGpXtQ>) e parte 2 (<http://bit.ly/1HGq0G9>) e *Things to Know About Python Super* parte 1 (O que devemos saber sobre *super* de Python, <http://bit.ly/1HGq1d4>), parte 2 (<http://bit.ly/1HGq1K7>) e parte 3 (<http://bit.ly/1HGq48I>). Os posts mais antigos usam a sintaxe de *super* de Python 2, mas continuam relevantes.

Li a primeira edição do livro *Object Oriented Analysis and Design, 3E* (Addison-Wesley, 2007) de Grady Booch e recomendo-o como uma introdução geral ao pensamento orientado a objetos, independentemente da linguagem de programação. É um dos poucos livros que abordam a herança múltipla sem preconceitos.

Ponto de vista

Pense nas classes de que você realmente precisa

A grande maioria dos programadores não escreve frameworks, mas aplicações. Mesmo aqueles que escrevem frameworks provavelmente gastam boa parte de seu tempo (se não a maior parte) escrevendo aplicações. Quando escrevemos aplicações, normalmente não precisamos implementar hierarquias de classes. No máximo, escrevemos classes que são subclasses de ABCs ou de outras classes oferecidas pelo framework. Como desenvolvedores de aplicações, é muito raro precisar escrever uma classe que será a superclasse de outra classe. As classes que implementamos quase sempre são folhas (isto é, folhas na árvore de herança).

Quando estiver trabalhando como desenvolvedor de aplicações, se você se vir criando hierarquias de classes com vários níveis, é provável que uma das opções a seguir se aplique:

- Você está reinventando a roda. Vá procurar um framework ou uma biblioteca que ofereça componentes que você possa reutilizar em sua aplicação.

- Você está usando um framework com design ruim. Vá procurar uma alternativa.
- Você está exagerando na engenharia. Lembre-se do *princípio KISS*.
- Você está entediado implementando aplicações e decidiu dar início a um novo framework. Parabéns e boa sorte!

Também é possível que todas as opções anteriores se apliquem à sua situação: você ficou entediado e decidiu reinventar a roda criando seu próprio framework, com excesso de engenharia e um design ruim, que está forçando você a implementar uma classe depois da outra para resolver problemas triviais. Espero que esteja se divertindo ou, pelo menos, sendo pago para isso.

Tipos embutidos malcomportados: bug ou feature?

Os tipos embutidos `dict`, `list` e `str` são blocos de construção essenciais do próprio Python, portanto devem ser eficientes – qualquer problema de desempenho nesses tipos teria impactos severos em quase tudo. É por isso que CPython adotou os atalhos que fazem os métodos de seus tipos embutidos se comportarem mal, não cooperando com os métodos sobrescritos por subclasses. Uma maneira possível de resolver esse dilema seria oferecer duas implementações para cada um desses tipos: uma “interna” – otimizada para uso do interpretador – e outra externa, facilmente extensível.

Mas espere um pouco, é isto que temos: `UserDict`, `UserList` e `UserString` não são tão rápidos quanto os tipos embutidos, mas são facilmente extensíveis. A abordagem pragmática adotada por CPython significa que também podemos usar, em nossas próprias aplicações, as implementações altamente otimizadas das quais é difícil herdar. Isso faz sentido, considerando que a necessidade de implementar um mapeamento, uma lista ou uma string especializados não é tão frequente, mas usamos `dict`, `list` e `str` no dia a dia. Só precisamos estar cientes do custo-benefício envolvido.

Herança em diferentes linguagens

Alan Kay cunhou o termo “orientado a objetos” e Smalltalk tinha apenas herança simples, embora haja variantes com várias formas de suporte a herança múltipla, incluindo os dialetos modernos Squeak e Pharo de Smalltalk, que têm suporte a *traits* – uma construção de linguagem que faz o papel de uma classe mixin, ao mesmo tempo evitando alguns dos problemas de herança múltipla.

A primeira linguagem popular a implementar herança múltipla foi C++, e houve tanto abuso no uso desse recurso que Java – cujo propósito era ser um substituto de C++ – foi projetada sem suporte a herança múltipla de implementação (ou

seja, sem classes mixin). Mas foi assim somente até Java 8 ter introduzido métodos default, que tornam as interfaces muito semelhantes às classes abstratas usadas para definir interfaces em C++ e em Python. Exceto que as interfaces em Java não podem ter estado – uma distinção fundamental. Depois de Java, provavelmente, a linguagem da JVM mais amplamente disseminada é Scala, que implementa traits. Outras linguagens com suporte a traits são as versões estáveis mais recentes de PHP e Groovy, além das linguagens em desenvolvimento Rust e Perl 6 – portanto pode-se dizer que traits são uma tendência neste momento (quando escrevi este livro).

Ruby oferece uma abordagem original à herança múltipla: não oferece suporte a ela, mas introduz mixins como um recurso da linguagem. Uma classe Ruby pode incluir um módulo em seu corpo, assim os métodos definidos no módulo tornam-se parte da implementação da classe. É uma forma “pura” de mixin, sem herança envolvida, e fica claro que uma mixin em Ruby não tem influência no tipo da classe em que ela é usada. Isso proporciona as vantagens das mixins, ao mesmo tempo que evita muitos de seus problemas usuais.

Duas linguagens recentes, que estão ganhando bastante impulso, limitam severamente a herança: Go e Julia. Go não tem nenhum tipo de herança, mas implementa interfaces de modo que lembra uma forma estática de duck typing (veja a seção “Ponto de vista” na página 390 para mais informações sobre esse assunto). Julia evita o termo “classes” e tem apenas “tipos”. Essa linguagem tem uma hierarquia de tipos, mas subtipos não podem herdar estruturas de dados internas – somente comportamentos –, e somente podem existir subtipos de tipos abstratos. Além disso, os métodos de Julia são implementados com despacho múltiplo (multiple dispatch) – uma forma mais avançada do mecanismo que vimos na seção “Funções genéricas com dispatch simples” na página 243.

CAPÍTULO 13

Sobrecarga de operadores: o jeito certo

Há algumas coisas sobre as quais me sinto dividido, por exemplo, sobrecarga de operadores. Deixei de lado a sobrecarga de operadores como uma escolha bem pessoal, pois havia visto muitas pessoas abusarem dela em C++.¹

— James Gosling
Criador de Java

A sobrecarga de operadores permite interoperação entre objetos definidos pelo usuário e operadores infixos como `+` e `|` ou com operadores unários como `-` e `~`. De modo mais amplo, chamada de função `(())`, acesso a atributos `(.)` e acesso a itens ou fatiamento `([])` também são operadores em Python, mas este capítulo discutirá operadores unários e infixos.

Na seção “Emulando tipos numéricos” na página 34 (Capítulo 1), vimos algumas implementações triviais de operadores em uma classe `Vector` básica. Os métodos `__add__` e `__mul__` do exemplo 1.2 foram escritos para mostrar como os métodos especiais dão suporte à sobrecarga de operadores, mas há problemas sutis em suas implementações que ignoramos naquele capítulo. Além disso, no exemplo 9.2, observamos que o método `Vector2d.__eq__` considera que o seguinte é `True`: `Vector(3, 4) == [3, 4]` – que pode ou não fazer sentido. Abordaremos essas questões neste capítulo.

Nas seções a seguir, discutiremos:

- Como Python trata operadores infixos com operandos de tipos diferentes.
- Uso de duck typing e de verificações explícitas de tipos para lidar com operandos de tipos variados.
- Como um método de operador infixos deve informar que não é capaz de tratar um operando.

¹ Fonte: “The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling” (A família de linguagens C: entrevista com Dennis Ritchie, Bjarne Stroustrup e James Gosling, http://www.gotw.ca/publications/c_family_interview.htm).

- O comportamento especial dos operadores de comparação rica (por exemplo, `==`, `>`, `<=` etc.).
- O tratamento-padrão dos operadores de atribuição combinada, como `+=`, e como sobrecarregá-los.

Básico da sobrecarga de operadores

A sobrecarga de operadores tem péssima reputação em alguns círculos. É um recurso da linguagem do qual pode-se abusar (e isso tem ocorrido), resultando em bugs, programadores perplexos e gargalos de desempenho surpreendentes. Contudo, se for bem usada, resulta em APIs agradáveis e códigos legíveis. Python apresenta um bom equilíbrio entre flexibilidade, usabilidade e proteção impondo algumas limitações:

- Não podemos sobrecarregar operadores para tipos embutidos.
- Não podemos criar novos operadores, somente sobrecarregar operadores existentes.
- Alguns operadores não podem ser sobrecarregados: `is`, `and`, `or`, `not` (mas os operadores bit a bit `&`, `|` e `~` podem).

No capítulo 10, já tínhamos um operador infixo em `Vector`: `==`, tratado pelo método `__eq__`. Neste capítulo, melhoraremos a implementação de `__eq__` para tratar operandos de tipos diferentes de `Vector`. Entretanto os operadores de comparação rica (`==`, `!=`, `>`, `<`, `>=`, `<=`) são casos especiais em sobrecarga de operadores, portanto começaremos sobrecarregando quatro operadores aritméticos em `Vector`: os operadores unários `-` e `+`, seguidos dos operadores infixos `+` e `*`.

Vamos começar pelo assunto mais fácil: operadores unários.

Operadores unários

Em *The Python Language Reference* (Guia de referência à linguagem Python), a seção “6.5. Unary arithmetic and bitwise operations” (Aritmética unária e operações bit a bit, <http://bit.ly/1JHV4bN>) lista três operadores unários, mostrados aqui com seus métodos especiais associados:

- (`__neg__`)

Negação aritmética unária. Se `x` é `-2` então `-x == 2`.

+ (`__pos__`)

Positivo aritmético unário. Normalmente, $x == +x$, mas há alguns casos em que isso não é verdade. Veja a caixa de texto “Quando x e $+x$ não são iguais” na página 422 se estiver curioso.

- (`__invert__`)

Inversão bit a bit de um inteiro, definido como $\sim x == -(x+1)$. Se x é 2 então $\sim x == -3$.

O capítulo “Data Model” (Modelo de dados, https://docs.python.org/3/reference/datamodel.html#object.__neg__) de *The Python Language Reference* também lista a função embutida `abs(...)` como um operador unário. O método especial associado é `__abs__`, como vimos antes na seção “Emulando tipos numéricos” na página 34.

É fácil dar suporte a operadores unários. Basta implementar o método especial apropriado, que receberá somente um argumento: `self`. Use a lógica que fizer sentido em sua classe, mas atenha-se à regra fundamental dos operadores: sempre devolva um novo objeto. Em outras palavras, não modifique `self`, mas crie e devolva uma nova instância de um tipo adequado.

No caso de `-` e `+`, o resultado provavelmente será uma instância da mesma classe de `self`; para `+`, devolver uma cópia de `self` é a melhor abordagem na maioria das vezes. Para `abs(...)`, o resultado deve ser um número escalar. Quanto a `~`, é difícil dizer o que seria um resultado sensato se você não estiver lidando com bits em um inteiro, mas em um ORM, pode fazer sentido devolver a negação de uma cláusula SQL `WHERE`, por exemplo.

Conforme prometido, implementaremos diversos operadores novos na classe `Vector` do capítulo 10. O exemplo 13.1 mostra o método `__abs__` que já tínhamos no exemplo 10.16, e os recém-adicionados métodos para operadores unários `__neg__` e `__pos__`.

Exemplo 13.1 – `vector_v6.py`: operadores unários `-` e `+` acrescentados ao exemplo 10.16

```
def __abs__(self):  
    return math.sqrt(sum(x * x for x in self))  
  
def __neg__(self):  
    return Vector(-x for x in self) ❶  
  
def __pos__(self):  
    return Vector(self) ❷
```

❶ Para calcular `-v`, crie um novo `Vector` com todos os componentes de `self` negados.

❷ Para calcular `+v`, crie um novo `Vector` com todos os componentes de `self`.

Lembre-se de que as instâncias de `Vector` são iteráveis e que `Vector.__init__` aceita um argumento iterável, portanto as implementações de `__neg__` e `__pos__` são compactas e descomplicadas.

- 7 Agora `one_third == +one_third` é `False`.
- 8 Inspecciona `+one_third`; há 28 dígitos após o '.' nesse caso.

O fato é que cada ocorrência da expressão `+one_third` produz uma nova instância de `Decimal` a partir do valor de `one_third`, mas usando a precisão do contexto aritmético atual.

O segundo caso em que `x != +x` pode ser encontrado na documentação de `collections.Counter` (<http://bit.ly/1JHV2E>). A classe `Counter` implementa vários operadores aritméticos, incluindo o operador infix `+` para somar os contadores de duas instâncias de `Counter`. Entretanto, por motivos práticos, a soma de `Counter` descarta qualquer item com uma contagem negativa ou igual a zero do resultado final. E o prefixo `+` é um atalho para somar um `Counter` vazio; sendo assim, ele produz um novo `Counter`, preservando somente os contadores maiores que zero. Veja o exemplo 13.3.

Exemplo 13.3 – O operador unário `+` gera um novo `Counter` sem contadores iguais a zero ou negativos

```
>>> ct = Counter('abracadabra')
>>> ct
Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
>>> ct['r'] = -3
>>> ct['d'] = 0
>>> ct
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 0, 'r': -3})
>>> +ct
Counter({'a': 5, 'b': 2, 'c': 1})
```

Voltamos agora para a nossa programação normal.

Sobrecarregando `+` para soma de vetores



A classe `Vector` é um tipo de sequência, e a seção “3.3.6. Emulating container types” (Emulando tipos para coleção, <http://bit.ly/1QOyDQY>) no capítulo “Data Model” (Modelo de dados) afirma que as sequências devem tratar o operador `+` para concatenação e `*` para repetição. Contudo, nesse caso, implementaremos `+` e `*` como operações matemáticas vetoriais, que são um pouco mais complexas, porém mais significativas para um tipo `Vector`.

Somar dois vetores euclidianos resulta em um novo vetor em que os componentes são as somas pareadas dos componentes de cada vetor. Para ilustrar:

```
>>> v1 = Vector([3, 4, 5])
>>> v2 = Vector([6, 7, 8])
>>> v1 + v2
Vector([9.0, 11.0, 13.0])
>>> v1 + v2 == Vector([3+6, 4+7, 5+8])
True
```

O que acontece se tentarmos somar duas instâncias de `Vector` de tamanhos diferentes? Poderíamos levantar um erro, mas, considerando as aplicações práticas (como recuperação de informações), é melhor preencher o `Vector` menor com zeros. Este é o resultado que queremos:

```
>>> v1 = Vector([3, 4, 5, 6])
>>> v3 = Vector([1, 2])
>>> v1 + v3
Vector([4.0, 6.0, 5.0, 6.0])
```

Dados esses requisitos básicos, a implementação de `__add__` é compacta e descomplicada, como mostra o exemplo 13.4.

Exemplo 13.4 – Método `Vector.add`, tomada #1

dentro da classe `Vector`

```
def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue=0.0) # ❶
    return Vector(a + b for a, b in pairs) # ❷
```

- ❶ `pairs` é um gerador que produzirá tuplas `(a, b)`, em que `a` vem de `self` e `b` vem de `other`. Se `self` e `other` tiverem tamanhos diferentes, `fillvalue` será usado para fornecer os valores ausentes ao iterável menor.
- ❷ Um novo `Vector` é criado a partir de uma expressão geradora que produz uma soma para cada item em `pairs`.

Observe como `__add__` devolve uma nova instância de `Vector`, sem afetar `self` ou `other`.



Métodos especiais que implementam operadores unários ou infixos jamais devem alterar seus operandos. Espera-se que expressões com operadores desse tipo produzam resultados criando novos objetos. Somente os operadores de atribuição combinada podem alterar o primeiro operando (`self`), conforme discutido na seção “Operadores de atribuição combinada” na página 438.

O exemplo 13.4 permite somar `Vector` a um `Vector2d`, e `Vector` a uma tupla ou a qualquer iterável que produza números, como comprova o exemplo 13.5.

Exemplo 13.5 – Vector.__add__ tomada #1 aceita também objetos que não são Vector

```
>>> v1 = Vector([3, 4, 5])
>>> v1 + (10, 20, 30)
Vector([13.0, 24.0, 35.0])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v1 + v2d
Vector([4.0, 6.0, 5.0])
```

As duas somas no exemplo 13.5 funcionam porque `__add__` usa `zip_longest(-)`, que pode consumir qualquer iterável, e a expressão geradora para criar o novo `Vector` simplesmente calcula `a + b` com os pares gerados por `zip_longest(-)`, portanto um iterável que produza itens numéricos funcionará.

Entretanto, se trocarmos os operandos de lugar (Exemplo 13.6), as somas de tipos diferentes falharão.

Exemplo 13.6 – Vector.__add__ tomada #1 falha com operandos diferentes de Vector à esquerda

```
>>> v1 = Vector([3, 4, 5])
>>> (10, 20, 30) + v1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "Vector") to tuple
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v2d + v1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2d' and 'Vector'
```

Para dar suporte a operações que envolvam objetos de tipos diferentes, Python implementa um mecanismo especial de despacho (dispatching) para os métodos especiais de operadores infixos. Dada uma expressão `a + b`, o interpretador executará os passos a seguir (veja também a figura 13.1):

1. Se `a` tiver `__add__`, chama `a.__add__(b)` e devolve o resultado, a menos que seja `NotImplemented`.
2. Se `a` não tiver `__add__`, ou sua chamada devolver `NotImplemented`, verifica se `b` tem `__radd__`, chama `b.__radd__(a)` e devolve o resultado, a menos que seja `NotImplemented`.
3. Se `b` não tiver `__radd__`, ou sua chamada devolver `NotImplemented`, levanta `TypeError` com uma mensagem *unsupported operand types* (tipos não aceitos de operandos).

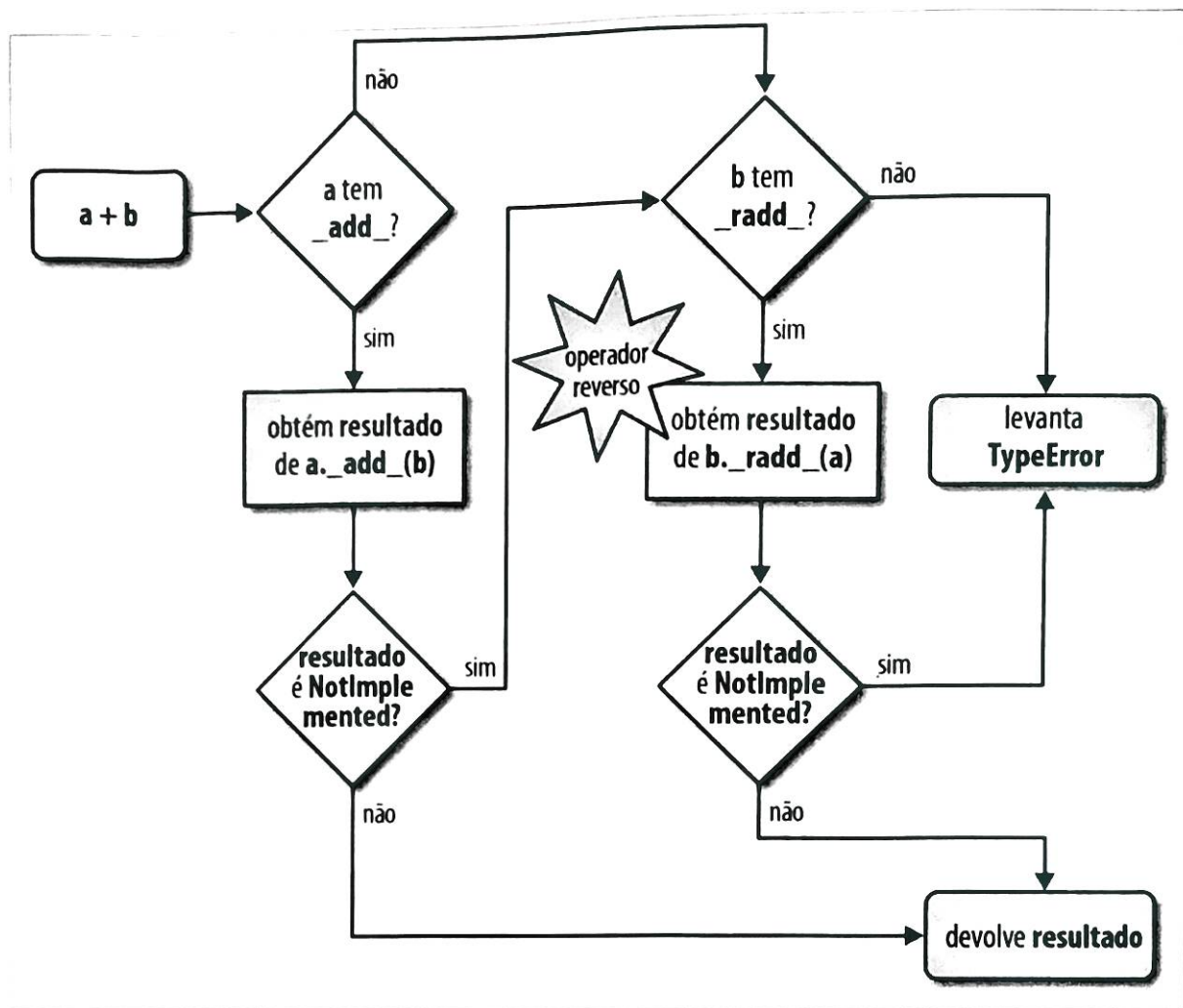


Figura 13.1 – Fluxograma para calcular `a + b` com `__add__` e `__radd__`.

O método `__radd__` é chamado de versão “refletida” (reflected) ou “reversa” (reversed) de `__add__`. Prefiro chamá-los de métodos especiais “reversos”.² Três dos revisores técnicos deste livro – Alex, Anna e Leo – disseram-me que gostam de pensar neles como métodos especiais da “direita” (right) porque são chamados no operando da direita. Independentemente de sua opção preferida, é daí que vem o prefixo “r” em `__radd__`, `__rsub__` e em métodos semelhantes.

Desse modo, para fazer as somas de tipos diferentes do exemplo 13.6 funcionarem, precisamos implementar o método `Vector.__radd__`, que Python chamará como alternativa se o operando da esquerda não implementar `__add__` ou se implementar, mas devolver `NotImplemented` para informar que não sabe como lidar com o operando da direita.

² A documentação de Python usa os dois termos. O capítulo “Data Model” (Modelo de dados, <https://docs.python.org/3/reference/datamodel.html>) usa “reflected” (refletido), mas a seção “9.1.2.2. Implementing the arithmetic operations” (Implementando as operações aritméticas, <http://bit.ly/1JHWP8W>) do módulo `numbers` menciona métodos “forward” (para frente) e “reversed” (reverso ou para trás), e acho essa terminologia melhor porque “forward” e “reversed” nomeiam claramente as duas direções, enquanto “reflected” não tem um oposto claro.



Não confunda `NotImplemented` com `NotImplementedError`. O primeiro, `NotImplemented`, é um valor singleton especial que o método especial de um operador infix deve devolver (via `return`) para dizer ao interpretador que não é capaz de lidar com um dado operando. Em comparação, `NotImplementedError` é uma exceção que um método vazio em uma classe abstrata pode levantar (via `raise`) para avisar que ele precisa ser sobrescrito nas subclasses.

O método `__radd__` mais simples possível e que funciona está no exemplo 13.7.

Exemplo 13.7 – Métodos `Vector.__add__` e `__radd__`

dentro da classe `Vector`

```
def __add__(self, other): # ❶
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    return Vector(a + b for a, b in pairs)

def __radd__(self, other): # ❷
    return self + other
```

❶ Sem alterações em `__add__` do exemplo 13.4; listado aqui porque `__radd__` o usa.

❷ `__radd__` simplesmente delega para `__add__`.

Frequentemente, `__radd__` pode ser simples assim: apenas chama o operador apropriado, delegando para `__add__` nesse caso. Isso se aplica a qualquer operador comutativo; `+` é comutativo quando lida com números ou com nossos vetores, mas não é comutativo quando concatena sequências em Python.

Os métodos do exemplo 13.4 funcionam com objetos `Vector` ou com qualquer iterável com itens numéricos, por exemplo, um `Vector2d`, uma tupla de inteiros ou um array de números de ponto flutuante. Porém, se um objeto não iterável for fornecido, `__add__` falhará com uma mensagem não muito útil, como mostra o exemplo 13.8.

Exemplo 13.8 – Método `Vector.__add__` precisa de um operando iterável

```
>>> v1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 328, in __add__
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
TypeError: zip_longest argument #2 must support iteration
```

Outra mensagem que não ajuda muito aparece quando um operando é iterável, mas seus itens não podem ser somados aos itens `float` de `Vector`. Veja o exemplo 13.9.

Exemplo 13.8 – Método `Vector.__add__` precisa de um iterável com itens numéricos

```
>>> v1 + 'ABC'
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "vector_v6.py", line 329, in `__add__`

return Vector(a + b for a, b in pairs)

File "vector_v6.py", line 243, in `__init__`

self._components = array(self.typecode, components)

File "vector_v6.py", line 329, in <genexpr>

return Vector(a + b for a, b in pairs)

`TypeError: unsupported operand type(s) for +: 'float' and 'str'`

Os problemas nos exemplos 13.8 e 13.9, na realidade, vão muito além de mensagens de erro obscuras: se o método especial de um operador não puder devolver um resultado válido devido à incompatibilidade de tipos, ele deve devolver `NotImplemented` em vez de levantar `TypeError`. Ao devolver `NotImplemented`, você deixa a porta aberta para que o implementador do tipo do outro operando realize a operação quando Python chamar o método reverso.

No espírito de duck typing, evitaremos testar o tipo do operando `other` ou o tipo de seus elementos. Capturaremos as exceções e devolveremos `NotImplemented`. Se o interpretador ainda não tiver invertido os operandos, ele tentará fazer isso. Se a chamada ao método reverso devolver `NotImplemented`, Python levantará `TypeError` com uma mensagem de erro padrão como “unsupported operand type(s) for +: *Vector* and *str*” (tipo(s) não suportado(s) de operando para +: *Vector* e *str*)

A implementação final dos métodos especiais para soma de `Vector` está no exemplo 13.10.

Exemplo 13.10 – `vector_v6.py`: métodos do operador + adicionados a `vector_v5.py` (Exemplo 10.16)

```
def __add__(self, other):
```

```
    try:
```

```
        pairs = itertools.zip_longest(self, other, fillvalue=0.0)
```

```
        return Vector(a + b for a, b in pairs)
```

```
    except TypeError:
```

```
        return NotImplemented
```

```
def __radd__(self, other):
```

```
    return self + other
```



Se o método de um operador infixo levantar uma exceção, o algoritmo de despacho do operador será abortado. No caso particular de `TypeError`, frequentemente será melhor capturá-lo e devolver `NotImplemented`. Isso permite que o interpretador tente chamar o método reverso do operador, que poderá tratar corretamente o cálculo com os operandos trocados, se forem de tipos diferentes.

A essa altura, sobrecarregamos o operador `+` de forma segura escrevendo `__add__` e `__radd__`. Vamos agora encarar outro operador infixo: `*`.

Sobrecarregando `*` para multiplicação por escalar

O que `Vector([1, 2, 3]) * x` quer dizer? Se `x` for um número, seria uma multiplicação por escalar, e o resultado seria um novo `Vector` com cada componente multiplicado por `x` – também conhecida como uma multiplicação elemento a elemento (element-wise multiplication):

```
>>> v1 = Vector([1, 2, 3])
>>> v1 * 10
Vector([10.0, 20.0, 30.0])
>>> 11 * v1
Vector([11.0, 22.0, 33.0])
```

Outro tipo de multiplicação envolvendo operandos `Vector` é o produto escalar (dot product) de dois vetores – ou multiplicação de matrizes, se você considerar um vetor como uma matriz $1 \times N$ e o outro como uma matriz $N \times 1$. A prática atual em NumPy e em bibliotecas semelhantes é não sobrecarregar `*` com esses dois significados, mas usar `*` somente para a multiplicação por escalar. Por exemplo, em NumPy, `numpy.dot()` calcula o produto escalar.³

De volta à nossa multiplicação escalar, novamente, começamos com os métodos `__mul__` e `__rmul__` mais simples possíveis:

```
# dentro da classe Vector
def __mul__(self, scalar):
    return Vector(n * scalar for n in self)

def __rmul__(self, scalar):
    return self * scalar
```

Esses métodos funcionam, exceto quando recebem operandos incompatíveis. O argumento `scalar` deve ser um número que, quando multiplicado por um `float`, produz

³ O sinal `@` pode ser usado como um operador infixo de produto escalar a partir de Python 3.5. Mais sobre ele na seção “O novo operador infixo `@` em Python 3.5” na página 432.

outro float (porque nossa classe `Vector` usa um array de números de ponto flutuante internamente). Desse modo, um número do tipo `complex` não servirá, mas o escalar pode ser um `int`, um `bool` (porque `bool` é uma subclasse de `int`) ou até mesmo uma instância de `fractions.Fraction`.

Poderíamos usar a mesma técnica de duck typing como fizemos no exemplo 13.10 e capturar um `TypeError` em `__mul__`, mas há outra maneira, mais explícita, que faz sentido nessa situação: *goose typing*. Usamos `isinstance()` para verificar o tipo de `scalar`, mas, em vez de deixar alguns tipos concretos fixos no código, fazemos a verificação com a `ABC numbers.Real`, que engloba todos os tipos de que precisamos e mantém nossa implementação aberta a futuros tipos numéricos que se declarem como subclasses reais ou *virtuais* da `ABC numbers.Real`. O exemplo 13.11 mostra um uso prático de *goose typing* – uma verificação explícita em relação a um tipo abstrato; consulte o repositório de código de *Python fluente* (<https://github.com/fluentpython/example-code>) para ver a listagem completa.



Como você deve se lembrar de seção “ABCs da biblioteca-padrão” na página 367, `decimal.Decimal` não está registrada como uma subclasse virtual de `numbers.Real`. Sendo assim, nossa classe `Vector` não tratará números `decimal.Decimal`.

Exemplo 13.11 – `vector_v7.py`: métodos do operador `*` adicionados

```
from array import array
import reprlib
import math
import functools
import operator
import itertools
import numbers # ❶

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    # muitos métodos foram omitidos na listagem do livro; veja vector_v7.py
    # em https://github.com/fluentpython/example-code ...

    def __mul__(self, scalar):
        if isinstance(scalar, numbers.Real): # ❷
            return Vector(n * scalar for n in self)
        else: # ❸
            return NotImplemented
```

```
def __rmul__(self, scalar):
    return self * scalar # ❹
```

- ❶ Importa o módulo `numbers` para verificação de tipo.
- ❷ Se `scalar` é uma instância de uma subclasse de `numbers.Real`, cria um novo `Vector` com os valores dos componentes multiplicados.
- ❸ Caso contrário, devolve `NotImplemented` para permitir que Python tente `__rmul__` no operando `scalar`.
- ❹ Nesse exemplo, `__rmul__` funciona bem simplesmente executando `self * scalar`, delegando ao método `__mul__`.

Com o exemplo 13.11, podemos multiplicar `Vectors` por valores escalares de tipos numéricos usuais e não tão usuais:

```
>>> v1 = Vector([1.0, 2.0, 3.0])
>>> 14 * v1
Vector([14.0, 28.0, 42.0])
>>> v1 * True
Vector([1.0, 2.0, 3.0])
>>> from fractions import Fraction
>>> v1 * Fraction(1, 3)
Vector([0.3333333333333333, 0.6666666666666666, 1.0])
```

Com a implementação de `+` e `*`, vimos os padrões mais comuns para implementar operadores infixos. As técnicas que descrevemos para `+` e `*` são aplicáveis a todos os operadores listados na tabela 13.1 (os operadores in-place serão discutidos na seção “Operadores de atribuição combinada” na página 438).

Tabela 13.1 – Nomes dos métodos para operadores infixos (os operadores in-place são usados em atribuição combinada; os operadores de comparação estão na tabela 13.2)

Operador	Normal (Forward)	Reverso (Reversed)	In-place	Descrição
<code>+</code>	<code>__add__</code>	<code>__radd__</code>	<code>__iadd__</code>	Adição ou concatenação
<code>-</code>	<code>__sub__</code>	<code>__rsub__</code>	<code>__isub__</code>	Subtração
<code>*</code>	<code>__mul__</code>	<code>__rmul__</code>	<code>__imul__</code>	Multiplicação ou repetição
<code>/</code>	<code>__truediv__</code>	<code>__rtruediv__</code>	<code>__itruediv__</code>	Divisão verdadeira
<code>//</code>	<code>__floordiv__</code>	<code>__rfloordiv__</code>	<code>__ifloordiv__</code>	Divisão pelo piso
<code>%</code>	<code>__mod__</code>	<code>__rmod__</code>	<code>__imod__</code>	Módulo
<code>divmod()</code>	<code>__divmod__</code>	<code>__rdivmod__</code>	<code>__idivmod__</code>	Devolve tupla com quociente da divisão pelo piso e módulo
<code>**</code> , <code>pow()</code>	<code>__pow__</code>	<code>__rpow__</code>	<code>__ipow__</code>	Exponencial*

Operador	Normal (Forward)	Reverso (Reversed)	In-place	Descrição
@	<code>__matmul__</code>	<code>__rmatmul__</code>	<code>__imatmul__</code>	Multiplicação de matrizes ^b
&	<code>__and__</code>	<code>__rand__</code>	<code>__iand__</code>	E (and) bit a bit
	<code>__or__</code>	<code>__ror__</code>	<code>__ior__</code>	Ou (or) bit a bit
^	<code>__xor__</code>	<code>__rxor__</code>	<code>__ixor__</code>	Ou exclusivo (xor) bit a bit
<<	<code>__lshift__</code>	<code>__rlshift__</code>	<code>__ilshift__</code>	Deslocamento para a esquerda (shift left) bit a bit
>>	<code>__rshift__</code>	<code>__rrshift__</code>	<code>__irshift__</code>	Deslocamento para a direita (shift right) bit a bit

^a `pow` aceita um terceiro argumento opcional, módulo: `pow(a, b, modulo)`, também aceito pelos métodos especiais quando chamado diretamente (por exemplo, `a.__pow__(b, modulo)`).

^b Novo em Python 3.5.

Os operadores de comparação rica são outra categoria de operadores infixos que usam um conjunto um pouco diferente de regras. Eles serão discutidos na próxima seção principal, “Operadores de comparação rica”, na página 433.

A caixa de texto opcional a seguir discute o operador `@` introduzido em Python 3.5 – que ainda não havia sido lançado quando escrevi este livro.

O novo operador infix `@` em Python 3.5

Python 3.4 não tem um operador infix para produto escalar. Entretanto, quando escrevi este livro, a versão pré-alfa de Python 3.5 já havia implementado a *PEP 465* — *A dedicated infix operator for matrix multiplication* (Um operador infix dedicado à multiplicação de matrizes, <https://www.python.org/dev/peps/pep-0465/>), deixando o sinal `@` disponível para essa operação (por exemplo, `a @ b` é o produto escalar de `a` e `b`). O operador `@` é tratado pelos métodos especiais `__matmul__`, `__rmatmul__` e `__imatmul__`, cujos nomes derivam de “matrix multiplication” (multiplicação de matrizes). Esses métodos não estão sendo usados em nenhum lugar na biblioteca-padrão atualmente, mas são reconhecidos pelo interpretador em Python 3.5, portanto a equipe de NumPy – assim como você e eu – pode implementar o operador `@` em tipos definidos pelo usuário. O parser também foi alterado para tratar o operador infix `@` (`a @ b` gera um erro de sintaxe em Python 3.4).

Somente por diversão, depois de compilar Python 3.5 a partir do código-fonte, pude implementar e testar o operador `@` para o produto escalar de `Vector`.

Estes são os testes simples que executei:

```
>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
```



```
>>> va @ vz == 38.0 # 1*5 + 2*6 + 3*7
```

```
True
```

```
>>> [10, 20, 30] @ vz
```

```
380.0
```

```
>>> va @ 3
```

```
Traceback (most recent call last):
```

```
...
```

```
TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
```

Veja o código dos métodos especiais relevantes:

```
class Vector:
```

```
    # muitos métodos foram omitidos na listagem do livro
```

```
    def __matmul__(self, other):
```

```
        try:
```

```
            return sum(a * b for a, b in zip(self, other))
```

```
        except TypeError:
```

```
            return NotImplemented
```

```
    def __rmatmul__(self, other):
```

```
        return self @ other
```

O código-fonte completo está no arquivo *vector_py3_5.py* no repositório de código de *Python fluente* (<https://github.com/fluentpython/example-code>).

Lembre-se de testá-lo em Python 3.5, senão você obterá um `SyntaxError`!

Operadores de comparação rica

O tratamento de operadores de comparação rica `==`, `!=`, `>`, `<`, `>=`, `<=` pelo interpretador Python é semelhante ao que acabamos de ver, mas difere em dois aspectos importantes:

- O mesmo conjunto de métodos é usado em chamadas de operadores diretos (forward) e reversos (reverse). As regras estão sintetizadas na tabela 13.2. Por exemplo, no caso de `==`, tanto a chamada direta quanto a reversa chamam `__eq__`, apenas trocando os argumentos de lugar; e uma chamada direta para `__gt__` é seguida de uma chamada reversa para `__lt__` com os argumentos trocados.
- No caso de `==` e de `!=`, se a chamada reversa falhar, Python irá comparar os IDs dos objetos em vez de levantar `TypeError`.

Tabela 13.2 – Operadores de comparação rica: métodos reversos chamados quando a chamada do método inicial devolver NotImplemented

Grupo	Operador infix	Chamada do método normal (forward)	Chamada do método reverso (reversed)	Alternativa
Igualdade	a == b	a.__eq__(b)	b.__eq__(a)	Devolve id(a) == id(b)
	a != b	a.__ne__(b)	b.__ne__(a)	Devolve not (a == b)
Ordenação	a > b	a.__gt__(b)	b.__lt__(a)	Levanta TypeError
	a < b	a.__lt__(b)	b.__gt__(a)	Levanta TypeError
	a >= b	a.__ge__(b)	b.__le__(a)	Levanta TypeError
	a <= b	a.__le__(b)	b.__ge__(a)	Levanta TypeError



Novo comportamento em Python 3

O passo alternativo para todos os operadores de comparação mudou em relação a Python 2. Para `__ne__`, Python 3 agora devolve o resultado negado de `__eq__`. Para os operadores de comparação de ordenação, Python 3 levanta `TypeError` com uma mensagem como `'unorderable types: int() < tuple()'` (tipos não podem ser ordenados: `int()` < `tuple()`). Em Python 2, essas comparações produziam resultados estranhos, considerando tipos e IDs de objetos de modo arbitrário. Contudo, realmente não faz sentido, por exemplo, comparar um `int` com um `tuple`, portanto levantar `TypeError` em casos como esse é uma verdadeira melhoria na linguagem.

Dadas essas regras, vamos rever e aperfeiçoar o comportamento do método `Vector.__eq__`, implementado da seguinte maneira em `vector_v5.py` (Exemplo 10.16):

```
class Vector:
    # muitas linhas foram omitidas
    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
```

Esse método gera os resultados do exemplo 13.12.

Exemplo 13.12 – Comparando um `Vector` com um `Vector`, um `Vector2d` e uma tupla

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb # ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
```

```
>>> v2d = Vector2d(1, 2)
>>> vc == v2d # ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3 # ❶
True
```

- ❶ Duas instâncias de `Vector` com componentes numéricos iguais são comparados como iguais.
- ❷ Um `Vector` e um `Vector2d` também são iguais se seus componentes forem iguais.
- ❸ Um `Vector` também é considerado igual a uma `tuple` ou a qualquer iterável com itens numéricos de valores iguais.

O último resultado do exemplo 13.12 provavelmente não é desejável. Realmente, não tenho uma regra rígida para esse caso; depende do contexto da aplicação. Porém o Zen do Python diz:

Diante de uma ambiguidade, não caia na tentação de adivinhar.

Liberdade em excesso na avaliação de operandos pode levar a resultados surpreendentes, e programadores detestam surpresas.

Com base no funcionamento do próprio Python, podemos ver que `[1,2] == (1, 2)` é `False`. Sendo assim, vamos ser conservadores e realizar algumas verificações de tipo. Se o segundo operando for uma instância de `Vector` (ou uma instância de uma subclasse de `Vector`), use a mesma lógica do `__eq__` atual. Caso contrário, devolva `NotImplemented` e deixe Python tratá-lo. Veja o exemplo 13.13.

Exemplo 13.13 – `vector_v8.py`: `__eq__` melhorado na classe `Vector`

```
def __eq__(self, other):
    if isinstance(other, Vector): ❶
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
    else:
        return NotImplemented ❷
```

- ❶ Se o operando `other` for uma instância de `Vector` (ou de uma subclasse de `Vector`), realize a comparação como antes.
- ❷ Caso contrário, devolva `NotImplemented`.

Se executar os testes do exemplo 13.12 com o novo `Vector.__eq__` do exemplo 13.13, você verá agora o que está no exemplo 13.14.

Exemplo 13.14 – Mesmas comparações do exemplo 13.12: o último resultado mudou

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb # ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d # ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3 # ❸
False
```

- ❶ Mesmo resultado anterior, conforme esperado.
- ❷ Mesmo resultado anterior, mas por quê? A explicação será dada a seguir.
- ❸ Resultado diferente; é isso que queríamos. Mas por que isso funciona? Continue lendo...

Entre os três resultados do exemplo 13.14, o primeiro não tem novidade, mas os dois últimos resultam de `__eq__` devolver `NotImplemented` no exemplo 13.13. Veja o que acontece passo a passo no exemplo com um `Vector` e um `Vector2d`:

1. Para avaliar `vc == v2d`, Python chama `Vector.__eq__(vc, v2d)`.
2. `Vector.__eq__(vc, v2d)` confirma que `v2d` não é um `Vector` e devolve `NotImplemented`.
3. Python obtém o resultado `NotImplemented`, portanto tenta executar `Vector2d.__eq__(v2d, vc)`.
4. `Vector2d.__eq__(v2d, vc)` transforma ambos os operandos em tuplas e os compara: o resultado é `True` (o código de `Vector2d.__eq__` está no exemplo 9.9).

Quanto à comparação entre `Vector` e `tuple` no exemplo 13.14, os passos são:

1. Para avaliar `va == t3`, Python chama `Vector.__eq__(va, t3)`.
2. `Vector.__eq__(va, t3)` confirma que `t3` não é um `Vector` e devolve `NotImplemented`.
3. Python obtém o resultado `NotImplemented`, portanto tenta executar `tuple.__eq__(t3, va)`.
4. `tuple.__eq__(t3, va)` não tem ideia do que seja um `Vector`, portanto devolve `NotImplemented`.
5. No caso especial de `==`, se a chamada reversa devolver `NotImplemented`, Python compara os IDs dos objetos como último recurso.

E o que dizer de !=? Não precisamos implementá-lo porque o comportamento alternativo para o método `__ne__` herdado de `object` é adequado para nós: quando `__eq__` está definido e não devolve `NotImplemented`, `__ne__` devolve esse resultado negado.

Em outras palavras, dados os mesmos objetos que usamos no exemplo 13.14, os resultados para != são consistentes:

```
>>> va != vb
False
>>> vc != v2d
False
>>> va != (1, 2, 3)
True
```

`__ne__` herdado de `object` funciona como o código a seguir – exceto que o original está implementado em C:⁴

```
def __ne__(self, other):
    eq_result = self == other
    if eq_result is NotImplemented:
        return NotImplemented
    else:
        return not eq_result
```



Bug na documentação de Python 3

Quando escrevi este livro, a documentação dos métodos de comparação rica (<https://docs.python.org/3/reference/datamodel.html>) afirmava o seguinte: “O fato de `x==y` ser verdadeiro não implica que `x!=y` seja falso. Da mesma forma, ao definir `__eq__()`, deve-se também definir `__ne__()` para que os operadores se comportem conforme esperado.” Isso era verdade em Python 2, mas em Python 3 não é um bom conselho, pois uma implementação default útil de `__ne__` é herdada da classe `object` e raramente será necessário sobrescrevê-la. O novo comportamento está documentado em *What’s New in Python 3.0* (O que há de novo em Python 3.0, <http://bit.ly/1C11zP5>) de Guido na seção “Operators And Special Methods” (Operadores e métodos especiais). O bug de documentação está registrado como *issue 4395* (<http://bugs.python.org/issue4395>).

Depois de ter abordado o essencial da sobrecarga de operadores infixos, vamos enfatizar uma categoria diferente de operadores: os operadores de atribuição combinada.

⁴ A lógica para `object.__eq__` e `object.__ne__` está na função `object_richcompare` em `Objects/typeobject.c` (<http://bit.ly/1C11uL7>) no código-fonte de CPython.

Operadores de atribuição combinada

Nossa classe `Vector` já aceita os operadores de atribuição combinada `+=` e `*=`. O exemplo 13.15 mostra-os em ação.

Exemplo 13.15 – A atribuição combinada funciona com alvos imutáveis criando novas instâncias e reassociando valores

```
>>> v1 = Vector([1, 2, 3])
>>> v1_alias = v1 # ❶
>>> id(v1) # ❷
4302860128
>>> v1 += Vector([4, 5, 6]) # ❸
>>> v1 # ❹
Vector([5.0, 7.0, 9.0])
>>> id(v1) # ❺
4302859904
>>> v1_alias # ❻
Vector([1.0, 2.0, 3.0])
>>> v1 *= 11 # ❼
>>> v1 # ❽
Vector([55.0, 77.0, 99.0])
>>> id(v1)
4302858336
```

- ❶ Cria um apelido (alias) para que possamos inspecionar o objeto `Vector([1, 2, 3])` mais tarde.
- ❷ Lembra o ID do `Vector` inicial associado a `v1`.
- ❸ Realiza uma soma combinada.
- ❹ O resultado é o esperado...
- ❺ ... mas um novo `Vector` foi criado.
- ❻ Inspeciona `v1_alias` para confirmar que o `Vector` não foi alterado.
- ❼ Realiza uma multiplicação combinada.
- ❽ Novamente, o resultado é o esperado, mas um novo `Vector` foi criado.

Se uma classe não implementa os operadores in-place listados na tabela 13.1, os operadores de atribuição combinada são apenas açúcar sintático: `a += b` será avaliado exatamente como `a = a + b`. Esse é o comportamento esperado para tipos imutáveis e, se você tiver `__add__`, o operador `+=` funcionará sem código adicional.

No entanto, se você implementar um método de operador in-place como `__iadd__`, esse método será chamado para calcular o resultado de `a += b`. Como diz o nome, espera-se que esses operadores mudem o operando da esquerda in-place em vez de criar um novo objeto como resultado.



Os métodos especiais in-place jamais devem ser implementados para tipos imutáveis como a nossa classe `Vector`. Isso é razoavelmente óbvio, mas vale a pena dizê-lo, de qualquer maneira.

Para mostrar o código de um operador in-place, estenderemos a classe `BingoCage` do exemplo 11.12 para implementar `__add__` e `__iadd__`.

Chamaremos a subclasse de `AddableBingoCage`. O exemplo 13.16 mostra o comportamento que queremos para o operador `+`.

Exemplo 13.16 – Uma nova instância de `AddableBingoCage` pode ser criada com:

```
>>> vowels = 'AEIOU'
>>> globe = AddableBingoCage(vowels) ❶
>>> globe.inspect()
('A', 'E', 'I', 'O', 'U')
>>> globe.pick() in vowels ❷
True
>>> len(globe.inspect()) ❸
4
>>> globe2 = AddableBingoCage('XYZ') ❹
>>> globe3 = globe + globe2
>>> len(globe3.inspect()) ❺
7
>>> void = globe + [10, 20] ❻
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'AddableBingoCage' and 'list'
```

- ❶ Cria uma instância `globe` com cinco itens (cada uma das vogais em `vowels`).
- ❷ Retira um item e confere se está em `vowels`.
- ❸ Confirma que `globe` foi reduzido a quatro itens.
- ❹ Cria uma segunda instância com três itens.
- ❺ Cria uma terceira instância somando as duas instâncias anteriores. A nova instância tem sete itens.

- ❹ A tentativa de somar um `AddableBingoCage` a uma `list` falha com `TypeError`. Essa mensagem de erro é gerada pelo interpretador Python quando nosso método `__add__` devolve `NotImplemented`.

Como um `AddableBingoCage` é mutável, o exemplo 13.17 mostra como ele funciona quando implementamos `__iadd__`.

Exemplo 13.17 – Um `AddableBingoCage` existente pode ser carregado com `+=` (continuação do exemplo 13.16)

```
>>> globe_orig = globe ❶
>>> len(globe.inspect()) ❷
4
>>> globe += globe2 ❸
>>> len(globe.inspect())
7
>>> globe += ['M', 'N'] ❹
>>> len(globe.inspect())
9
>>> globe is globe_orig ❺
True
>>> globe += 1 ❻
Traceback (most recent call last):
...
TypeError: right operand in += must be 'AddableBingoCage' or an iterable
```

- ❶ Cria um apelido para que possamos verificar a identidade do objeto mais tarde.
- ❷ `globe` tem quatro itens aqui.
- ❸ Uma instância de `AddableBingoCage` pode receber itens de outra instância da mesma classe.
- ❹ O operando à direita de `+=` também pode ser qualquer iterável.
- ❺ Ao longo desse exemplo, `globe` sempre referenciou o objeto `globe_orig`.
- ❻ Tentar somar um não iterável a um `AddableBingoCage` falha com uma mensagem de erro apropriada.

Observe que o operador `+=` é mais liberal que `+` em relação ao segundo operando. Com `+`, queremos que ambos os operandos sejam do mesmo tipo (`AddableBingoCage`, nesse caso) porque, se aceitarmos tipos diferentes, isso pode causar confusão quanto ao tipo do resultado. Com `+=`, a situação é mais clara: o objeto à esquerda é atualizado in-place, portanto não há dúvidas sobre o tipo do resultado.



Validei o comportamento contrastante de `+` e de `+=` observando como o tipo embutido `list` funciona. Ao escrever `my_list + x`, você só pode concatenar uma `list` a outra `list`, mas se escrever `my_list += x`, é possível estender a `list` da esquerda com itens de qualquer iterável `x` do lado direito. Isso é consistente com o funcionamento do método `list.extend()`: ele aceita qualquer argumento iterável.

Agora que temos clareza quanto ao comportamento desejado para `AddableBingoCage`, podemos ver sua implementação no exemplo 13.18.

Exemplo 13.18 – `bingoaddable.py`: `AddableBingoCage` estende `BingoCage` para tratar `+` e `+=`

```
import itertools ❶

from tombola import Tombola
from bingo import BingoCage

class AddableBingoCage(BingoCage): ❷
    def __add__(self, other):
        if isinstance(other, Tombola): ❸
            return AddableBingoCage(self.inspect() + other.inspect()) ❹
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, Tombola):
            other_iterable = other.inspect() ❺
        else:
            try:
                other_iterable = iter(other) ❻
            except TypeError: ❼
                self_cls = type(self).__name__
                msg = "right operand in += must be {!r} or an iterable"
                raise TypeError(msg.format(self_cls))
        self.load(other_iterable) ❽
        return self ❾
```

❶ A PEP 8 — *Style Guide for Python Code* (Guia de estilo para código Python, <https://www.python.org/dev/peps/pep-0008/#imports>) recomenda declarar importações da biblioteca-padrão antes das importações de seus próprios módulos.

❷ `AddableBingoCage` estende `BingoCage`.

❸ Nosso `__add__` só funcionará com uma instância de `Tombola` como segundo operando.

- ④ Recupera itens de `other` se for uma instância de `Tombola`.
- ⑤ Caso contrário, tenta obter um iterador para `other`.⁵
- ⑥ Se isso falhar, levanta uma exceção explicando o que o usuário deve fazer. Quando possível, mensagens de erro devem orientar explicitamente o usuário para a solução.
- ⑦ Se chegamos até aqui, podemos carregar `other_iterable` em `self`.
- ⑧ Muito importante: os métodos especiais de atribuição combinada devem devolver `self`.

Podemos resumir toda a ideia de operadores in-place comparando as instruções `return` que produzem resultados em `__add__` e `__iadd__` no exemplo 13.18:

`__add__`

O resultado é produzido pela chamada ao construtor `AddableBingoCage` para criar uma nova instância.

`__iadd__`

O resultado é produzido devolvendo `self`, depois de ter sido modificado.

Para concluir esse exemplo, uma última observação sobre o exemplo 13.18: por design, `__radd__` não foi implementado em `AddableBingoCage`, por não ser necessário. O método direto (forward) `__add__` lidará com operandos à direita somente do mesmo tipo, portanto, se Python estiver tentando calcular `a + b`, em que `a` é um `AddableBingoCage` mas `b` não é, devolvemos `NotImplemented` – talvez a classe de `b` possa fazer essa operação funcionar. Porém se a expressão for `b + a` e `b` não for um `AddableBingoCage`, e `NotImplemented` for devolvido, será melhor deixar Python desistir e levantar `TypeError` porque não sabemos lidar com `b`.



Em geral, se um método de operador infix normal (por exemplo, `__mul__`) for projetado para funcionar somente com operandos do mesmo tipo de `self`, será inútil implementar o método reverso correspondente (por exemplo, `__rmul__`) porque ele, por definição, só será chamado quando lidarmos com um operando de um tipo diferente.

Com isso, concluímos nossa exploração da sobrecarga de operadores em Python.

⁵ A função embutida `iter` será discutida no próximo capítulo. Nesse caso, eu poderia ter usado `tuple(other)` e funcionaria, mas ao custo de criar uma nova `tuple` quando tudo que o método `.load(...)` precisa fazer é uma iteração pelo seu argumento.

Resumo do capítulo

Iniciamos este capítulo analisando algumas restrições impostas por Python à sobrecarga de operadores: não é permitido sobrecarregar operadores nos tipos embutidos, e a sobrecarga está limitada aos operadores existentes, com exceção de alguns (`is`, `and`, `or`, `not`).

Colocamos a mão na massa com os operadores unários, implementando `__neg__` e `__pos__`. Em seguida, vieram os operadores infixos, começando por `+`, tratado pelo método `__add__`. Vimos que é esperado que os operadores unários e infixos gerem resultados criando novos objetos e jamais devem alterar seus operandos. Para aceitar operações com outros tipos, devolvemos o valor especial `NotImplemented` – e não uma exceção – permitindo ao interpretador tentar novamente trocando os operandos de lugar e chamando o método especial reverso desse operador (por exemplo, `__radd__`). O algoritmo que Python usa para tratar operadores infixos está sintetizado no fluxograma da figura 13.1.

Misturar tipos diferentes de operandos significa que precisamos identificar um operando que não sabemos tratar. Neste capítulo, fizemos isso de duas maneiras: no estilo de duck typing, simplesmente fomos em frente e tentamos executar a operação, capturando uma exceção `TypeError` caso ela ocorresse; depois em `__mul__`, fizemos isso com um teste explícito usando `isinstance`. Há prós e contras nessas abordagens: duck typing é mais flexível, mas verificações explícitas de tipo são mais previsíveis. Quando usamos `isinstance`, tivemos o cuidado de evitar testes com uma classe concreta e usamos a `ABC numbers.Real`: `isinstance(scalar, numbers.Real)`. Esse é um bom compromisso entre flexibilidade e proteção, pois tipos existentes ou definidos pelo usuário no futuro poderão ser declarados como subclasses reais ou virtuais de uma `ABC`, como vimos no capítulo 11.

O próximo assunto que discutimos foram os operadores de comparação rica. Implementamos `==` com `__eq__` e descobrimos que Python oferece uma implementação conveniente de `!=` no `__ne__` herdado da classe-base `object`. O modo como Python avalia esses operadores, juntamente com `>`, `<`, `>=` e `<=`, é um pouco diferente, com uma lógica especial para escolher o método reverso e um tratamento alternativo para `==` e `!=`, que nunca gera erros porque Python compara os IDs dos objetos como último recurso.

Na última seção, voltamos nossa atenção aos operadores de atribuição combinada. Vimos que Python os trata por default como uma combinação de um operador simples, seguido da atribuição, ou seja: `a += b` é avaliado exatamente como `a = a + b`. Isso sempre cria um novo objeto, portanto funciona para tipos mutáveis e imutáveis. Para objetos mutáveis, podemos implementar métodos especiais in-place como `__iadd__` para `+=` e alterar o valor do operando da esquerda. Para mostrar isso em funcionamento, deixamos de lado a classe imutável `Vector` e trabalhamos na implementação

de uma subclasse de `BingoCage` que trata `+=` a fim de adicionar itens à coleção aleatória, semelhante ao modo como o tipo embutido `list` trata `+=` como um atalho para o método `list.extend()`. Enquanto fizemos isso, discutimos como `+` tende a ser mais rígido que `+=` em relação aos tipos aceitos. Para tipos de sequência, `+` normalmente exige que ambos os operandos sejam do mesmo tipo, enquanto `+=` em geral aceita qualquer iterável como operando da direita.

Leituras complementares

A sobrecarga de operadores é uma área da programação Python em que testes com `isinstance` são comuns. Em geral, as bibliotecas devem aproveitar-se da tipagem dinâmica – para serem mais flexíveis – evitando testes explícitos de tipo e apenas tentando executar operações e, em seguida, tratando as exceções, de modo a abrir a porta para trabalhar com objetos que implementem as operações necessárias, independentemente de seus tipos. Mas as ABCs de Python permitem uma forma mais rigorosa de duck typing, apelidada de “goose typing” por Alex Martelli, que, frequentemente, é útil ao escrever códigos para sobrecarga de operadores. Portanto, se você pulou o capítulo 11, não deixe de lê-lo.

A principal referência para os métodos especiais de operadores é o capítulo “Data Model” (Modelo de dados, <https://docs.python.org/3/reference/datamodel.html>). É a fonte canônica, mas atualmente está contaminada com o bug evidente mencionado no aviso “Bug na documentação de Python 3”, aconselhando que “ao definir `__eq__()`, deve-se também definir `__ne__()`”. Na verdade, o `__ne__` herdado da classe `object` em Python 3 atende à grande maioria dos requisitos, portanto implementar `__ne__` raramente será necessário na prática. Outra leitura relevante na documentação de Python é “9.1.2.2. Implementing the arithmetic operations” (Implementando as operações aritméticas, <http://bit.ly/1JHWP8W>) no módulo `numbers` de *The Python Standard Library* (Biblioteca-
-Padrão de Python).

Uma técnica relacionada são as funções genéricas, tratadas pelo decorador `@singledispatch` em Python 3 (seção “Funções genéricas com dispatch simples” na página 243). No livro *Python Cookbook*, 3E (O’Reilly)⁶, de David Beazley e Brian K. Jones, “Recipe 9.20. Implementing Multiple Dispatch with Function Annotations” (Implementar múltiplos despachos com anotações de função) usa um pouco de metaprogramação avançada – envolvendo uma metaclasses – para implementar um despacho baseado em tipos, com anotações de função. A segunda edição de *Python Cookbook* de Martelli, Ravenscroft e Ascher tem uma receita interessante (2.13, de Erik Max Francis) que mostra como sobrecarregar o operador `<<` para emular a sintaxe de

6 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

`iostream` de C++ em Python. Ambos os livros têm outros exemplos com sobrecarga de operadores; selecionei apenas duas receitas dignas de nota.

A função `functools.total_ordering` é um decorador de classe (aceito em Python 2.7 e em versões mais recentes) que gera métodos automaticamente para todos os operadores de comparação rica em qualquer classe que defina pelo menos dois deles. Consulte a documentação do módulo `functools` (<http://bit.ly/1C12IWF>).

Se estiver curioso a respeito de despacho de métodos de operadores em linguagens com tipagem dinâmica, duas leituras essenciais são “A Simple Technique for Handling Multiple Polymorphism” (Uma técnica simples para tratar polimorfismo múltiplo, <http://bit.ly/1FVhejw>) de Dan Ingalls (membro da equipe original de Smalltalk) e “Arithmetic and Double Dispatching in Smalltalk-80” (Aritmética e despacho duplo em Smalltalk-80, <http://bit.ly/1QrnuuD>) de Kurt J. Hebel e Ralph Johnson (Johnson ficou famoso como um dos autores do livro *Design Patterns*⁷ original). Ambos os artigos oferecem esclarecimentos profundos quanto ao poder do polimorfismo em linguagens com tipagem dinâmica como Smalltalk, Python e Ruby. Python não usa despacho duplo para tratar operadores, como descrito nesses artigos. O algoritmo de Python, que usa operadores diretos (forward) e reversos (reverse), é mais fácil de ser tratado pelas classes definidas pelo usuário do que o despacho duplo, mas exige um tratamento especial pelo interpretador. Em comparação, o despacho duplo clássico é uma técnica genérica que você pode usar em Python ou em qualquer linguagem orientada a objetos fora do contexto específico de operadores infixos e, de fato, Ingalls, Hebel e Johnson usam exemplos muito diferentes para descrevê-lo.

O artigo “The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling” (A família de linguagens C: entrevista com Dennis Ritchie, Bjarne Stroustrup e James Gosling, http://www.gotw.ca/publications/c_family_interview.htm), da qual extraí a epígrafe deste capítulo, e dois outros trechos de código da seção “Ponto de vista” na página 446, apareceram em *Java Report*, 5(7), julho de 2000 e em *C++ Report*, 12(7), julho/agosto de 2000. São leituras sensacionais se você estiver interessado em design de linguagens de programação.

Ponto de vista

Sobrecarga de operadores: prós e contras

James Gosling, autor da citação do início deste capítulo, tomou a decisão consciente de deixar a sobrecarga de operadores de fora quando projetou Java. Na mesma entrevista – “The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling” (A família de linguagens C: entrevista

⁷ N.T.: Edição brasileira publicada com o título “Padrões de projeto” (Bookman).

com Dennis Ritchie, Bjarne Stroustrup e James Gosling”, <http://bit.ly/1C12T4t>) –, ele diz o seguinte:

Provavelmente, cerca de 20% a 30% da população pensa que sobrecarga de operadores é criação do demônio; alguém fez algo com sobrecarga de operadores que simplesmente os irritou porque usaram, por exemplo, + para inserção em lista, e isso deixou a vida muito, muito confusa. Muitos desses problemas se originam do fato de haver apenas cerca de meia dúzia de operadores que podem ser sobrecarregados de forma sensata e, apesar disso, há milhares ou milhões de operadores que as pessoas gostariam de definir – sendo assim, você precisa escolher um, e, com frequência, as escolhas entram em conflito com seu senso de intuição.

Guido van Rossum escolheu o meio-termo para tratar sobrecarga de operadores: não deixou a porta aberta aos usuários para que criassem operadores novos e arbitrários como `<=>` ou `:-)`, o que evita uma Torre de Babel de operadores personalizados e permite que o parser de Python seja simples. Python também não permite que você sobrecarregue os operadores de tipos embutidos: outra limitação que promove legibilidade e um desempenho previsível.

Gosling prossegue dizendo:

Daí há uma comunidade de aproximadamente 10% que realmente usou sobrecarga de operadores de forma apropriada e que se interessa por ela, e para quem ela é realmente importante; é um grupo formado quase exclusivamente por pessoas que trabalham com números, em que a notação é muito importante para a intuição das pessoas, pois elas têm uma intuição sobre o que + significa, e a capacidade de dizer “a + b”, em que a e b são números complexos ou matrizes ou outro dado, realmente faz sentido.

O aspecto referente à notação não pode ser menosprezado. Veja um exemplo ilustrativo da área de finanças. Em Python, você pode calcular juros compostos usando uma fórmula como esta:

```
interest = principal * ((1 + rate) ** periods - 1)
```

A mesma notação funciona, independentemente dos tipos numéricos envolvidos. Desse modo, se você estiver fazendo um trabalho financeiro sério, poderá garantir que `periods` seja um `int`, enquanto `rate`, `interest` e `principal` sejam números exatos – instâncias da classe `decimal.Decimal` de Python –, e essa fórmula funcionará exatamente como está escrita.

Em Java, porém, se mudar de `float` para `BigDecimal` para obter uma precisão controlada, você não poderá mais usar operadores infixos, pois eles funcionam somente com tipos primitivos. Esta é a mesma fórmula implementada para funcionar com números `BigDecimal` em Java:

```
BigDecimal interest = principal.multiply(BigDecimal.ONE.add(rate))  
                                .pow(periods).subtract(BigDecimal.ONE));
```


Está claro que operadores infixos deixam as fórmulas mais legíveis, pelo menos, para a maioria de nós.⁸ A sobrecarga de operadores é necessária para dar suporte à notação de operadores infixos para tipos não primitivos. Ter sobrecarga de operadores em uma linguagem de alto nível, fácil de usar, provavelmente foi um motivo essencial para a incrível penetração de Python em computação científica nos últimos anos.

É claro que também há vantagens em não permitir a sobrecarga de operadores em uma linguagem. É provavelmente uma decisão sensata para linguagens de sistemas de baixo nível, nas quais desempenho e proteção são muito importantes. A recente linguagem Go seguiu o caminho de Java quanto a esse aspecto e não aceita sobrecarga de operadores.

Porém os operadores sobrecarregados, quando usados de forma sensata, deixam o código mais fácil de ler e de escrever. É um ótimo recurso em uma linguagem de alto nível moderna.

Uma espiada na avaliação preguiçosa (lazy evaluation)

Se você observar atentamente o traceback do exemplo 13.9, verá evidências da avaliação preguiçosa (*lazy evaluation*) de expressões geradoras. O exemplo 13.19 apresenta o mesmo traceback, agora com comentários numerados.

Exemplo 13.19 – Igual ao exemplo 13.9

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs) # ❶
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components) # ❷
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs) # ❸
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

- ❶ A chamada a `Vector` recebe uma expressão geradora como seu argumento `components`. Até aí, nenhum problema.
- ❷ A `genexp components` é passada para o construtor `array`. No construtor `array`, Python tenta fazer iteração pela `genexp`, causando a avaliação do primeiro item `a + b`. É nesse momento que `TypeError` ocorre.

⁸ Meu amigo Mario Domenech Goulart, um core developer do compilador CHICKEN para Scheme (<http://www.call-cc.org/>), provavelmente não concordará com isso.

③ A exceção se propaga até a chamada ao construtor `Vector`, quando é informada. Isso mostra como a expressão geradora é avaliada o mais tardar possível, e não no ponto em que é definida no código-fonte.

Em comparação, se o construtor `Vector` fosse chamado como `Vector([a + b for a, b in pairs])`, a exceção ocorreria exatamente aí, pois a list comprehension tentaria criar uma list para ser passada como argumento para a chamada a `Vector()`. O corpo de `Vector.__init__` jamais seria alcançado.

O capítulo 14 abordará as expressões geradoras em detalhes, mas não queria que essa demonstração acidental de sua natureza preguiçosa passasse despercebida.

PARTE V

Controle de fluxo

CAPÍTULO 14

Iteráveis, iteradores e geradores

Quando vejo padrões em meus programas, considero isso sinal de um problema. A forma de um programa deve refletir apenas o problema que ele precisa resolver. Qualquer outra regularidade no código é sinal, pelo menos para mim, de que estou usando abstrações que não são poderosas o suficiente – muitas vezes, de que estou gerando manualmente as expansões de alguma macro que eu deveria escrever.¹

— Paul Graham

Hacker de Lisp e investidor em startups

Iteração é fundamental para processamento de dados. Ao percorrer conjuntos de dados que não cabem na memória, precisamos ter como acessar os itens de modo *lazy* (*preguiçoso*), isto é, um item de cada vez, por demanda. É disso que trata o padrão `Iterator` (Iterador). Este capítulo mostra como esse padrão está integrado à linguagem Python de modo que você jamais precisará implementá-lo manualmente.

Python não tem macros como Lisp (a linguagem favorita de Paul Graham), portanto abstrair o padrão `Iterator` exigiu mudanças na linguagem: a palavra reservada `yield` foi acrescentada em Python 2.2 (2001).² A palavra `yield` permite a construção de geradores que funcionam como iteradores.



Todo gerador é um iterador: geradores implementam completamente a interface `Iterator`. Porém um iterador – conforme definido no livro da GoF – recupera itens de uma coleção, enquanto um gerador pode produzir itens “do nada”. É por isso que o gerador da sequência de Fibonacci é um exemplo comum: uma série infinita de números não pode ser armazenada em uma coleção. Mas saiba que a comunidade Python usa as palavras *iterador* e *gerador* como sinônimos na maioria das vezes.

Python 3 usa geradores em vários lugares. Até a função embutida `range()` atualmente devolve um objeto gerador ou similar, em vez de listas completas, como antes.

¹ De “Revenge of the Nerds” (Vingança dos nerds, <http://www.paulgraham.com/icad.html>), um post de blog.

² Usuários de Python 2.2 podiam usar `yield` com a diretiva `from __future__ import generators`; `yield` tornou-se disponível por padrão em Python 2.3.

Se precisar criar uma `list` a partir de `range`, você precisa ser explícito (por exemplo, `list(range(100))`).

Toda coleção em Python é *iterável*, e os iteradores são usados internamente para dar suporte a:

- laços `for`;
- construção e extensão de tipos para coleção;
- percorrer arquivos-texto linha a linha em um laço;
- `list`, `dict` e `set` comprehensions;
- desempacotamento de tuplas;
- desempacotamento de parâmetros com `*` em chamadas de função.

Este capítulo aborda os tópicos a seguir:

- como a função embutida `iter(...)` é usada internamente para tratar objetos iteráveis;
- como implementar o padrão `Iterator` clássico em Python;
- operação de uma função geradora em detalhes, com descrições linha a linha;
- como o padrão `Iterator` clássico pode ser substituído por uma função ou expressão geradora;
- como aproveitar as funções geradoras de propósito geral da biblioteca-padrão;
- uso do novo comando `yield from` para combinar geradores;
- um estudo de caso: uso de funções geradoras em um utilitário de conversão de banco de dados projetado para trabalhar com grandes massas de dados;
- por que geradores e corrotinas se parecem, mas, na verdade, são muito diferentes e não devem ser confundidos.

Começaremos estudando como a função `iter(...)` faz qualquer sequência ser iterável.

Sentence tomada #1: uma sequência de palavras

Começaremos nossa exploração de iteráveis implementando uma classe `Sentence`: forneça uma string com um texto ao seu construtor e você poderá iterar palavra por palavra. A primeira versão implementa o protocolo de sequência, e é iterável porque todas as sequências são iteráveis, como já vimos, mas agora veremos exatamente por quê.

O exemplo 14.1 mostra uma classe `Sentence` que extrai palavras de um texto pelo índice.

Exemplo 14.1 – sentence.py: uma Sentence como uma sequência de palavras

```
import re
import reprlib

RE_WORD = re.compile('\w+')

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text) ❶

    def __getitem__(self, index):
        return self.words[index] ❷

    def __len__(self): ❸
        return len(self.words)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text) ❹
```

- ❶ `re.findall` devolve uma lista com todos os casamentos³ da expressão regular, sem sobreposição, como uma lista de strings.
- ❷ `self.words` armazena o resultado de `.findall`, portanto simplesmente devolvemos a palavra no índice especificado.
- ❸ Para completar o protocolo de sequência, implementamos `__len__` – mas ele não é necessário para deixar um objeto iterável.
- ❹ `reprlib.repr` é uma função utilitária para gerar representações string abreviadas de estruturas de dados que sejam muito grandes.⁴

Por padrão, `reprlib.repr` limita a string gerada a 30 caracteres. Observe a sessão de console no exemplo 14.2 para ver como `Sentence` é usada.

Exemplo 14.2 – Testando a iteração em uma instância de Sentence

```
>>> s = Sentence('"The time has come," the Walrus said,') # ❶
>>> s
Sentence('"The time ha... Walrus said,') # ❷
>>> for word in s: # ❸
...     print(word)
```

3 Nota do autor/revisor: citando o Aurelio Marinho Jargas em seu excelente livro *Expressões Regulares*, uma abordagem divertida, publicado pela editora Novatec: “Casamento aqui não é juntar os trapos, mas sim o ato de bater, conferir, igualar, encontrar, encaixar, equiparar. É como em um caixa 24 horas, em que você só retirará o dinheiro se a sua senha digitada casar com aquela já cadastrada no banco.”

4 Usamos `reprlib` pela primeira vez na seção “Vector tomada #1: compatível com `Vector2d`” na página 319.

```
The  
time  
has  
come  
the  
Walrus  
said  
>>> list(s) # ❶  
['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

- ❶ Uma sentença é criada a partir de uma string.
- ❷ Observe a saída de `__repr__` com `...`, gerada por `reprlib.repr`.
- ❸ Instâncias de `Sentence` são iteráveis; veremos o porquê em breve.
- ❹ Por serem iteráveis, objetos `Sentence` podem ser usados como entrada para criar listas e outros tipos iteráveis.

Nas páginas a seguir, desenvolveremos outras classes `Sentence` que passarão nos testes do exemplo 14.2. Entretanto a implementação do exemplo 14.1 é diferente de todas as demais porque também é uma sequência, portanto você pode obter as palavras pelo índice:

```
>>> s[0]  
'The'  
>>> s[5]  
'Walrus'  
>>> s[-1]  
'said'
```

Todo programador Python sabe que sequências são iteráveis. Agora veremos exatamente por quê.

Por que sequências são iteráveis: a função `iter`

Sempre que o interpretador precisa iterar por um objeto `x`, ele chama `iter(x)` automaticamente.

A função embutida `iter`:

1. Verifica se o objeto implementa `__iter__` e o chama para obter um iterador.
2. Se `__iter__` não estiver implementado, mas `__getitem__` estiver, Python criará um iterador que tentará acessar os itens em sequência, começando no índice 0 (zero).
3. Se isso falhar, Python levantará `TypeError`, normalmente dizendo “C object is not iterable” (objeto `C` não é iterável), em que `C` é a classe do objeto-alvo.

É por isso que qualquer sequência em Python é iterável: todas elas implementam `__getitem__`. De fato, as sequências-padrões também implementam `__iter__`, e a sua deve implementar também, pois o tratamento especial de `__getitem__` existe por causa de compatibilidade com versões anteriores, e talvez não esteja mais presente no futuro (embora ainda não fosse obsoleto quando escrevi este livro).

Conforme mencionado na seção “Python curte sequências” na página 355, essa é uma forma extrema de duck typing: um objeto é considerado iterável não só quando implementa o método especial `__iter__`, mas também quando implementa `__getitem__`, desde que `__getitem__` aceite chaves `int`, começando em 0.

Na abordagem com goose typing, a definição de um iterável é mais simples, mas não tão flexível: um objeto é considerado iterável se implementar o método `__iter__`. Não há necessidade de subclasses nem de registros, pois `abc.Iterable` implementa `__subclasshook__`, como vimos na seção “Gansos podem se comportar como patos” na página 385. Veja uma demonstração:

```
>>> class Foo:
...     def __iter__(self):
...         pass
...
>>> from collections import abc
>>> issubclass(Foo, abc.Iterable)
True
>>> f = Foo()
>>> isinstance(f, abc.Iterable)
True
```

Porém observe que nossa classe `Sentence` inicial não passa pelo teste `issubclass(Sentence, abc.Iterable)`, apesar de ser iterável na prática.



Em Python 3.4, a maneira mais exata de verificar se um objeto `x` é iterável é chamar `iter(x)` e tratar uma exceção `TypeError` se não for. Isso é mais preciso que usar `isinstance(x, abc.Iterable)` porque `iter(x)` também considera o método legado `__getitem__`, enquanto a `ABC Iterable` não o faz.

Verificar explicitamente se um objeto é iterável pode não valer a pena se, logo após a verificação, você fizer uma iteração pelo objeto. A final de contas, quando tentamos fazer uma iteração em um objeto não iterável, a exceção levantada por Python é bem clara: `TypeError: 'C' object is not iterable` (`TypeError`: objeto ‘C’ não é iterável). Se puder fazer algo melhor que simplesmente levantar `TypeError`, faça isso em um bloco `try/except` em vez de fazer uma verificação explícita. A verificação explícita pode fazer sentido se você estiver guardando o objeto para iterar sobre ele mais tarde; nesse caso, capturar o erro com antecedência pode ser conveniente.

A próxima seção torna explícito o relacionamento entre iteráveis e iteradores.

Iteráveis versus iteradores

A partir da explicação na seção “Por que sequências são iteráveis: a função `iter`” na página 453, podemos extrapolar e fazer uma definição:

iterável

Qualquer objeto a partir do qual a função embutida `iter` pode obter um iterador. Objetos que implementem um método `__iter__` que devolva um *iterador* são iteráveis. Sequências sempre são iteráveis, assim como objetos que implementem um método `__getitem__` que aceite índices a partir de 0.

É importante ser claro sobre o relacionamento entre iteráveis e iteradores: Python obtém iteradores a partir de iteráveis.

Veja um laço `for` simples fazendo uma iteração por uma `str`. A `str` `'ABC'` é o iterável nesse caso. Há um iterador em operação neste exemplo, mas ele não está visível no código:

```
>>> s = 'ABC'
>>> for char in s:
...     print(char)
...
A
B
C
```

Se não existisse o comando `for` e tivéssemos que emular o seu funcionamento manualmente com um laço `while`, teríamos que escrever o seguinte:

```
>>> s = 'ABC'
>>> it = iter(s) # ❶
>>> while True:
...     try:
...         print(next(it)) # ❷
...     except StopIteration: # ❸
...         del it # ❹
...         break # ❺
...
A
B
C
```

❶ Cria um iterador `it` a partir do iterável.

- ❷ Chama `next` repetidamente no iterador para obter o próximo item.
- ❸ O iterador levanta `StopIteration` quando não há mais itens.
- ❹ Libera a referência a `it` – o objeto iterador é descartado.
- ❺ Sai do laço.

`StopIteration` informa que o iterador esgotou. Essa exceção é tratada internamente em laços `for` e em outros contextos de iteração, como `list comprehensions`, desempacotamento de tuplas etc.

A interface-padrão de um iterador tem dois métodos:

`__next__`

Devolve o próximo item disponível, levantando `StopIteration` quando não houver mais itens.

`__iter__`

Devolve `self`; permite que iteradores sejam usados nos lugares em que se espera um iterável, por exemplo, em um laço `for`.

Isso está formalizado na ABC `collections.abc.Iterator`, que define o método abstrato `__next__` e é subclasse de `Iterable` – na qual o método abstrato `__iter__` está definido. Veja a figura 14.1.

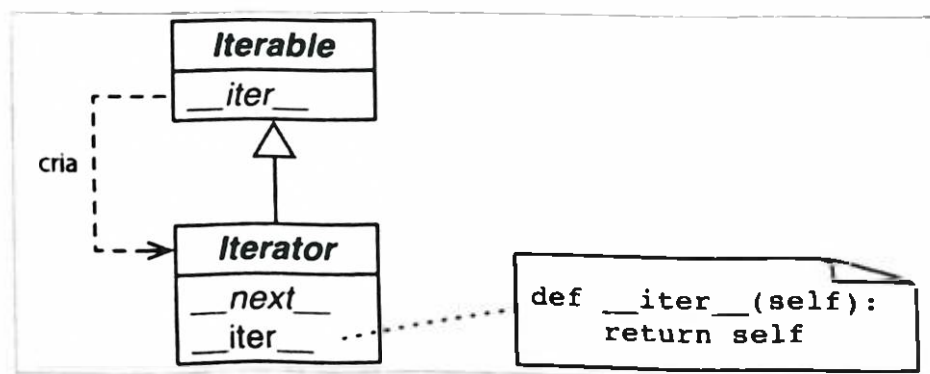


Figura 14.1 – As ABCs `Iterable` e `Iterator`. Métodos em itálico são abstratos. Um `Iterable.__iter__` concreto tem que devolver uma nova instância de `Iterator`. Um `Iterator` concreto deve implementar `__next__`. O método `Iterator.__iter__` simplesmente devolve a própria instância.

A ABC `Iterator` implementa `__iter__` executando `return self`. Isso permite que um iterador seja usado em qualquer lugar onde um iterável é exigido. O código-fonte de `abc.Iterator` está no exemplo 14.3.

Exemplo 14.3 – classe `abc.Iterator`, extraída de `Lib/_collections_abc.py` (<http://bit.ly/1C14QOi>)

```
class Iterator(Iterable):
    __slots__ = ()
    @abstractmethod
    def __next__(self):
        'Return the next item from the iterator. When exhausted, raise StopIteration'
        raise StopIteration
    def __iter__(self):
        return self
    @classmethod
    def __subclasshook__(cls, C):
        if cls is Iterator:
            if (any("__next__" in B.__dict__ for B in C.__mro__) and
                any("__iter__" in B.__dict__ for B in C.__mro__)):
                return True
        return NotImplemented
```



O método abstrato da ABC `Iterator` é `it.__next__()` em Python 3 e `it.next()` em Python 2. Como sempre, evite chamar diretamente os métodos especiais. Use apenas `next(it)`: essa função embutida faz o que é certo em Python 2 e 3.

O código-fonte do módulo `Lib/types.py` (<https://hg.python.org/cpython/file/3.4/Lib/types.py>) em Python 3.4 tem o seguinte comentário:

```
# Iterators in Python aren't a matter of type but of protocol. A large
# and changing number of builtin types implement *some* flavor of
# iterator. Don't check the type! Use hasattr to check for both
# "__iter__" and "__next__" attributes instead.5
```

De fato, é exatamente isso que o método `__subclasshook__` da ABC `abc.Iterator` faz (veja o exemplo 14.3).



Levando em conta o conselho de `Lib/types.py` e a lógica implementada em `Lib/_collections_abc.py`, a melhor maneira de verificar se um objeto `x` é um iterador é chamar `isinstance(x, abc.Iterator)`. Graças a `Iterator.__subclasshook__`, esse teste funciona, mesmo que a classe de `x` não seja uma subclasse real nem virtual de `Iterator`.

De volta à nossa classe `Sentence` do exemplo 14.1, você pode ver claramente como o iterador é criado por `iter(...)` e consumido por `next(...)` usando o console de Python:

5 N.T.: “Iteradores em Python não são uma questão de tipo, mas de protocolo. Uma quantidade grande e variável de tipos embutidos implementa “alguma” variante de iterador. Não verifique o tipo! Em vez disso, use `hasattr` para conferir a presença dos dois atributos `__iter__` e `__next__`”.

```

>>> s3 = Sentence('Pig and Pepper') # ❶
>>> it = iter(s3) # ❷
>>> it # doctest: +ELLIPSIS
<iterator object at 0x...>
>>> next(it) # ❸
'Pig'
>>> next(it)
'and'
>>> next(it)
'Pepper'
>>> next(it) # ❹
Traceback (most recent call last):
...
StopIteration
>>> list(it) # ❺
[]
>>> list(iter(s3)) # ❻
['Pig', 'and', 'Pepper']

```

- ❶ Cria uma sentença `s3` com três palavras.
- ❷ Obtém um iterador a partir de `s3`.
- ❸ `next(it)` busca a próxima palavra.
- ❹ Não há mais palavras, portanto o iterador levanta uma exceção `StopIteration`.
- ❺ Depois de esgotado, um iterador torna-se inútil.
- ❻ Para percorrer a sentença novamente, é preciso construir um novo iterador.

Como os únicos métodos exigidos de um iterador são `__next__` e `__iter__`, não há como verificar se há itens restantes, a não ser chamando `next()` e tratando `StopIteration`. Além disso, não é possível “reiniciar” um iterador. Se precisar recomeçar, chame `iter(...)` no iterável a partir do qual o iterador foi criado anteriormente. Chamar `iter(...)` no próprio iterador não ajudará, pois – como mencionamos – `Iterator.__iter__` é implementado devolvendo `self`, portanto isso não reiniciará um iterador esgotado.

Para encerrar esta seção, apresentamos uma definição de *iterador*:

iterador

Qualquer objeto que implemente o método `__next__`, sem argumentos, que devolva o próximo item de uma série ou levante `StopIteration` quando não houver mais itens. Os iteradores em Python também implementam o método `__iter__`, portanto também são *iteráveis*.

A primeira versão de `Sentence` era iterável graças ao tratamento especial dado pela função embutida `iter(...)` às sequências. Vamos implementar agora o protocolo-padrão de iteráveis.

Sentence tomada #2: um iterador clássico

A próxima classe `Sentence` será criada de acordo com o padrão de projeto `Iterator` clássico, seguindo o esquema do livro da GoF. Observe que não usaremos Python idiomático, como as próximas refatorações deixarão bem claro. Mas serve para deixar explícito o relacionamento entre a coleção iterável e o objeto iterador.

O exemplo 14.4 mostra uma implementação de uma `Sentence` iterável porque implementa o método especial `__iter__`, que cria e devolve um `SentenceIterator`. É assim que o padrão de projeto `Iterator` é descrito no livro *Design Patterns*⁶ original.

Implementaremos dessa maneira aqui somente para deixar clara a distinção fundamental entre um iterável e um iterador e como eles estão relacionados.

Exemplo 14.4 – `sentence_iter.py`: `Sentence` implementada com o padrão `Iterator`

```
import re
import reprlib

RE_WORD = re.compile('\w+')

class Sentence:
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self): ❶
        return SentenceIterator(self.words) ❷

class SentenceIterator:
    def __init__(self, words):
        self.words = words ❸
        self.index = 0 ❹

    def __next__(self):
        try:
            word = self.words[self.index] ❺
```

6 N.T.: Edição brasileira publicada com o título “Padrões de projeto” (Bookman).

```

except IndexError:
    raise StopIteration() ❹
self.index += 1 ❺
return word ❻
def __iter__(self): ❼
    return self

```

- ❶ O método `__iter__` é o único acréscimo à implementação anterior de `Sentence`. Essa versão não tem `__getitem__` para deixar claro que a classe é iterável porque implementa `__iter__`.
- ❷ `__iter__` atende ao protocolo de iteráveis ao instanciar e devolver um iterador.
- ❸ `SentenceIterator` armazena uma referência à lista de palavras.
- ❹ `self.index` é usado para determinar a próxima palavra a ser buscada.
- ❺ Obtém a palavra na posição `self.index`.
- ❻ Se não há uma palavra em `self.index`, levanta `StopIteration`.
- ❼ Incrementa `self.index`.
- ❽ Devolve a palavra.
- ❾ Implementa `self.__iter__`.

O código do exemplo 14.4 passa nos testes do exemplo 14.2.

Note que implementar `__iter__` em `SentenceIterator` não é realmente necessário para esse exemplo funcionar, mas é o certo a fazer: espera-se que os iteradores implementem tanto `__next__` quanto `__iter__`, e fazer isso faz nosso iterador passar no teste `issubclass(SentenceIterator, abc.Iterator)`. Se tivéssemos criado `SentenceIterator` como subclasse de `abc.Iterator`, teríamos herdado o método concreto `abc.Iterator.__iter__`.

O código do exemplo 14.4 é bastante trabalhoso (pelo menos para nós, programadores Python preguiçosos). Observe como a maior parte do código de `SentenceIterator` lida com o gerenciamento do estado interno do iterador. Em breve, veremos como deixá-lo mais conciso. Antes disso, porém, faremos um rápido desvio para discutir um atalho de implementação que pode ser tentador, mas é errado.

Fazer de `Sentence` um iterador: péssima ideia

Uma causa comum de erros na criação de iteráveis e iteradores é confundi-los. Para esclarecer: iteráveis têm um método `__iter__` que sempre instancia um novo iterador. Iteradores implementam um método `__next__` que devolve itens individuais e um método `__iter__` que devolve `self`.

Desse modo, iteradores também são iteráveis, mas iteráveis não são iteradores.

Pode ser tentador implementar `__next__` além de `__iter__` na classe `Sentence`, fazendo cada instância de `Sentence` ser, ao mesmo tempo, um iterável e um iterador de si mesma. Porém essa é uma péssima ideia. Também é um antipadrão comum, de acordo com Alex Martelli, que tem muita experiência com revisões de código em Python.

A seção “Applicability”⁷ (Aplicabilidade) do padrão de projeto `Iterator` do livro da *GoF* afirma o seguinte:

Use o padrão `Iterator` para:

- acessar o conteúdo de um objeto agregado sem expor sua representação interna;
- permitir que objetos agregados sejam percorridos por vários clientes;
- oferecer uma interface uniforme para percorrer diferentes estruturas agregadas (ou seja, suportar iteração polimórfica).

Para “permitir que objetos agregados sejam percorridos por vários clientes”, deve ser possível obter vários iteradores independentes a partir da mesma instância de iterável, e cada iterador deve manter seu próprio estado interno, portanto uma implementação apropriada do padrão exige que cada chamada a `iter(my_iterable)` crie um novo iterador independente. É por isso que precisamos da classe `SentenceIterator` nesse exemplo.



Um iterável jamais deve atuar como um iterador sobre si mesmo. Em outras palavras, iteráveis devem implementar `__iter__`, mas não `__next__`. Por outro lado, por conveniência, iteradores devem ser iteráveis. O método `__iter__` de um iterador deve simplesmente devolver `self`.

Agora que o padrão `Iterator` clássico foi devidamente demonstrado, podemos deixá-lo de lado. A próxima seção apresenta uma implementação mais idiomática de `Sentence`.

Sentence tomada #3: uma função geradora

Uma implementação pythônica da mesma funcionalidade usa uma função geradora para substituir a classe `SequenceIterator`. Uma explicação apropriada sobre a função geradora está logo depois do exemplo 14.5.

Exemplo 14.5 – `sentence_gen.py`: `Sentence` implementada com uma função geradora

```
import re
import replib

RE_WORD = re.compile('\w+')
```

7 Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 259.

```

class Sentence:
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for word in self.words: ❶
            yield word ❷
        return ❸

# pronto! ❹

```

- ❶ Faz uma iteração por `self.word`.
- ❷ Produz a `word` atual.
- ❸ Esse `return` não é realmente necessário; a função pode simplesmente terminar e retornar automaticamente. De qualquer modo, uma função geradora não levanta `StopIteration`: ela simplesmente sai quando acaba de produzir valores.⁸
- ❹ Não há necessidade de uma classe separada de iterador!

Aqui, novamente, temos uma implementação diferente de `Sentence`, que passa nos testes do exemplo 14.2.

De volta ao código de `Sentence` no exemplo 14.4, `__iter__` chamou o construtor `SentenceIterator` para criar um iterador e devolvê-lo. Agora, no exemplo 14.5, o iterador é, na verdade, um objeto gerador, criado automaticamente quando o método `__iter__` é chamado, pois `__iter__` é uma função geradora.

Uma explicação completa sobre as funções geradoras vem a seguir.

Como funciona uma função geradora

Qualquer função Python que tenha a palavra reservada `yield` em seu corpo é uma função geradora: uma função que, quando chamada, devolve um objeto gerador. Em outras palavras, uma função geradora é uma fábrica (factory) de geradores.

⁸ Ao analisar esse código, Alex Martelli sugeriu que o corpo desse método poderia ser simplesmente `return iter(self.words)`. Ele está certo, é claro: o resultado de chamar `__iter__` também seria um iterador, como deveria. No entanto usei um laço `for` com `yield` nesse caso para introduzir a sintaxe de uma função geradora, que será discutida em detalhes na próxima seção.



A única sintaxe que distingue uma função comum de uma função geradora é o fato de a última ter uma palavra reservada `yield` em algum lugar de seu corpo. Algumas pessoas argumentaram que uma nova palavra reservada, por exemplo, `gen`, deveria ter sido definida para declarar funções geradoras no lugar de `def`, mas Guido não concordou. Seus argumentos estão na PEP 255 — *Simple Generators*⁹ (Geradores simples, <https://www.python.org/dev/peps/pep-0255/>).

⁹ Às vezes, acrescento um prefixo ou sufixo `gen` ao nomear funções geradoras, mas essa não é uma prática comum. Além disso, você não poderá fazer isso se estiver implementando um iterável, é claro: o método especial necessário precisa se chamar `__iter__`.

Veja a função mais simples possível que serve para mostrar o comportamento de um gerador:⁹

```
>>> def gen_123(): # ❶
...     yield 1 # ❷
...     yield 2
...     yield 3
...
>>> gen_123 # doctest: +ELLIPSIS
<function gen_123 at 0x...> # ❸
>>> gen_123() # doctest: +ELLIPSIS
<generator object gen_123 at 0x...> # ❹
>>> for i in gen_123(): # ❺
...     print(i)
1
2
3
>>> g = gen_123() # ❻
>>> next(g) # ❼
1
>>> next(g)
2
>>> next(g)
3
>>> next(g) # ❽
Traceback (most recent call last):
...
StopIteration
```

❶ Qualquer função em Python que contenha a palavra reservada `yield` é uma função geradora.

⁹ Agradeço a David Kwast por sugerir esse exemplo.

- ❷ Normalmente, o corpo de uma função geradora tem um laço, mas não necessariamente; nesse caso, simplesmente repete `yield` três vezes.
- ❸ Ao observar mais de perto, vemos que `gen_123` é um objeto-função.
- ❹ Mas, quando é chamada, `gen_123()` devolve um objeto gerador.
- ❺ Geradores são iteradores que produzem os valores das expressões passadas para `yield`.
- ❻ Para inspecionar mais de perto, atribuímos o objeto gerador a `g`.
- ❼ Como `g` é um iterador, chamar `next(g)` busca o próximo item produzido por `yield`.
- ❽ Quando o corpo da função termina, o objeto gerador levanta um `StopIteration`.

Uma função geradora cria um objeto gerador que engloba o corpo da função. Quando chamamos `next(...)` no objeto gerador, a execução avança para o próximo `yield` no corpo da função, e a chamada a `next(...)` resulta no valor que foi produzido pelo `yield` onde o corpo da função foi suspenso. Por fim, quando o corpo da função retorna, o objeto gerador que o engloba levanta `StopIteration`, de acordo com o protocolo de `Iterator`.



Acho útil ser rigoroso ao falar dos resultados obtidos de um gerador: digo que um gerador *produz* (yields) valores. Mas é confuso dizer que um gerador “devolve” ou “retorna” (returns) valores. Funções devolvem valores. Chamar uma função geradora devolve um gerador. Um gerador produz valores. Um gerador não “devolve” valores da maneira usual: a instrução `return` no corpo de uma função geradora faz `StopIteration` ser levantada pelo objeto gerador.^(*)

(*) Antes de Python 3.3, fornecer um valor com o comando `return` em uma função geradora era um erro. Agora isso é permitido, mas `return` ainda faz uma exceção `StopIteration` ser levantada. Quem fez a chamada pode recuperar o valor de retorno por meio do objeto de exceção. Contudo isso é relevante somente quando uma função geradora é usada como corrotina, como veremos na seção “Devolvendo um valor a partir de uma corrotina” na página 528.

O exemplo 14.6 deixa mais explícita a interação entre um laço `for` e o corpo da função.

Exemplo 14.6 – Uma função geradora que exibe mensagens quando é executada

```
>>> def gen_AB(): # ❶
...     print('start')
...     yield 'A'    # ❷
...     print('continue')
...     yield 'B'    # ❸
...     print('end.') # ❹
... 
```

```
>>> for c in gen_AB(): # ❶
...     print('-->', c) # ❷
...
start ❸
--> A ❹
continue ❺
--> B ❻
end. ❼
>>> ❽
```

- ❶ A função geradora é definida como qualquer função, mas usa `yield`.
- ❷ A primeira chamada implícita a `next()` no laço `for` em ❶ exibirá `'start'` e parará no primeiro `yield`, produzindo o valor `'A'`.
- ❸ A segunda chamada implícita a `next()` no laço `for` exibirá `'continue'` e parará no segundo `yield`, produzindo o valor `'B'`.
- ❹ A terceira chamada a `next()` exibirá `'end.'` e prosseguirá para o final do corpo da função, fazendo o objeto gerador levantar `StopIteration`.
- ❺ Para iterar, a lógica interna do comando `for` faz o equivalente a `g = iter(gen_AB())` para obter um objeto gerador e, depois, `next(g)` a cada iteração.
- ❻ O bloco com o laço exibe `-->` e um valor devolvido por `next(g)`. Mas essa saída será vista somente depois da saída das chamadas a `print` na função geradora.
- ❼ A string `'start'` aparece como resultado de `print('start')` no corpo da função geradora.
- ❽ `yield 'A'` no corpo da função geradora produz o valor `'A'` consumido pelo laço `for`, que é atribuído à variável `c` e resulta na saída `--> A`.
- ❾ A iteração continua com uma segunda chamada a `next(g)`, avançando o corpo da função geradora de `yield 'A'` para `yield 'B'`. O texto `continue` é exibido por causa do segundo `print` no corpo da função geradora.
- ❿ `yield 'B'` produz o valor `'B'` consumido pelo laço `for`, que é atribuído à variável `c` do laço, de modo que o laço exibe `--> B`.
- ⓫ A iteração continua com uma terceira chamada a `next(it)`, avançando para o final do corpo da função. O texto `end.` aparece na saída por causa do terceiro `print` no corpo da função geradora.
- ⓬ Quando o corpo da função geradora atinge o final, o objeto gerador levanta `StopIteration`. O mecanismo do laço `for` captura essa exceção e o laço termina de forma limpa.

Espero que agora esteja claro como `Sentence.__iter__` no exemplo 14.5 funciona: `__iter__` é uma função geradora que, quando chamada, cria um objeto gerador; esse objeto implementa a interface de iteradores, de modo que a classe `SentenceIterator` não é mais necessária.

A segunda versão de `Sentence` no exemplo 14.5 é muito menor que a primeira, mas não é tão preguiçosa (lazy) quanto poderia ser. Atualmente, ser preguiçoso é considerado uma qualidade, pelo menos em linguagens de programação e APIs. Uma implementação lazy adia a produção de valores até o último instante possível. Isso economiza memória e também pode evitar processamentos desnecessários.

Criaremos uma classe `Sentence` preguiçosa a seguir.

Sentence tomada #4: uma implementação lazy

A interface de `Iterator` foi concebida para ser lazy: `next(my_iterator)` produz um item de cada vez. O oposto de lazy (preguiçoso) é eager (ávido); avaliação preguiçosa (lazy evaluation) e avaliação ávida (eager evaluation), na verdade, são termos técnicos na teoria de linguagens de programação.

Nossas implementações de `Sentence` até agora não foram lazy porque `__init__` cria uma lista de forma ávida (eager) com todas as palavras do texto, associando-a ao atributo `self.words`. Isso implica processar todo o texto, e a lista pode usar tanta memória quanto o próprio texto (provavelmente mais; depende de quantos caracteres não usados em palavras existem no texto). A maior parte desse trabalho será em vão se o usuário iterar somente pelas primeiras palavras.

Sempre que estiver usando Python 3 e se perguntar “Há uma maneira lazy de fazer isso?”, com frequência, a resposta será “Sim”.

A função `re.finditer` é uma versão lazy de `re.findall`; em vez de uma lista, ela devolve um gerador que produz instâncias de `re.MatchObject` sob demanda. Se houver muitos casamentos, `re.finditer` economizará bastante memória. Com esta função, nossa terceira versão de `Sentence` agora é lazy; produzirá a próxima palavra somente quando for necessário. O código está no exemplo 14.7.

Exemplo 14.7 – `sentence_gen2.py`: `Sentence` implementada com uma função geradora que chama a função geradora `re.finditer`

```
import re
import replib

RE_WORD = re.compile('\w+')
```

```
class Sentence:
    def __init__(self, text):
        self.text = text ❶

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for match in RE_WORD.finditer(self.text): ❷
            yield match.group() ❸
```

❶ Não é preciso ter uma lista `words`.

❷ `finditer` cria um iterador para os casamentos de `RE_WORD` em `self.text`, produzindo instâncias de `MatchObject`.

❸ `match.group()` extrai o texto que casou da instância de `MatchObject`.

As funções geradoras são um atalho fantástico, mas o código pode ficar menor ainda com uma expressão geradora.

Sentence tomada #5: uma expressão geradora

Funções geradoras simples, como aquela da classe `Sentence` que acabamos de ver (Exemplo 14.7), podem ser substituídas por uma expressão geradora.

Uma expressão geradora pode ser entendida como uma versão lazy de uma list comprehension: ela não cria uma lista de modo ávido (eager), mas devolve um gerador que produzirá itens por demanda de modo preguiçoso (lazy). Em outras palavras, se uma list comprehension é uma fábrica de listas, uma expressão geradora é uma fábrica de geradores.

O exemplo 14.8 apresenta uma demo rápida de uma expressão geradora, comparando-a a uma list comprehension.

Exemplo 14.8 – A função geradora `gen_AB` é usada por uma list comprehension e depois por uma expressão geradora

```
>>> def gen_AB(): # ❶
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()] # ❷
```



```

start
continue
end.
>>> for i in res1: # ❸
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB()) # ❹
>>> res2 # ❺
<generator object <genexpr> at 0x10063c240>
>>> for i in res2: # ❻
...     print('-->', i)
...
start
--> AAA
continue
--> BBB
end.

```

- ❶ Essa é a mesma função `gen_AB` do exemplo 14.6.
- ❷ A list comprehension itera avidamente pelos itens produzidos pelo objeto gerador resultante da chamada a `gen_AB()`: 'A' e 'B'. Observe a saída nas próximas linhas: `start, continue, end.`
- ❸ Esse laço `for` itera pela lista `res1` construída pela list comprehension.
- ❹ A expressão geradora devolve `res2`. Chamamos `gen_AB()`, mas essa chamada devolveu um gerador, que não foi consumido aqui.
- ❺ `res2` é um objeto gerador.
- ❻ Somente quando o laço `for` itera por `res2`, o corpo de `gen_AB` é realmente executado. Cada iteração do laço `for` chama `next(res2)` implicitamente, avançando `gen_AB` para o próximo `yield`. Observe a saída de `gen_AB` com a saída de `print` no laço `for`.

Portanto uma expressão geradora produz um gerador, e podemos usá-lo para reduzir mais ainda o código da classe `Sentence`. Veja o exemplo 14.9.

Exemplo 14.9 – `sentence_genexp.py`: `Sentence` implementada com uma expressão geradora

```

import re
import replib

RE_WORD = re.compile('\w+')

```

```
class Sentence:
    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        return (match.group() for match in RE_WORD.finditer(self.text))
```

A única diferença em relação ao exemplo 14.7 é o método `__iter__`, que, nesse caso, não é uma função geradora (não tem `yield`), mas usa uma expressão geradora para criar um gerador e então o devolve. O resultado final é o mesmo: quem chama `__iter__` obtém um objeto gerador.

As expressões geradoras são açúcar sintático; elas sempre podem ser substituídas por funções geradoras, mas às vezes são mais convenientes. A próxima seção aborda o uso de expressões geradoras.

Expressões geradoras: quando usá-las

Usei várias expressões geradoras ao implementar a classe `Vector` no exemplo 10.16. Os métodos `__eq__`, `__hash__`, `__abs__`, `angle`, `angles`, `format`, `__add__` e `__mul__` usam expressões geradoras. Em todos aqueles métodos, uma *list comprehension* também funcionaria, ao custo de usar mais memória para armazenar os valores intermediários em listas.

No exemplo 14.9, vimos que uma expressão geradora é um atalho sintático para criar um gerador sem definir e chamar uma função. Por outro lado, funções geradoras são muito mais flexíveis: você pode implementar uma lógica complexa com vários comandos e pode até mesmo usá-las como *corrotinas* (veja o capítulo 16).

Em casos mais simples, uma expressão geradora será suficiente, e é mais legível, como mostram os exemplos de `Vector`.

Minha regra geral para escolher a sintaxe a ser usada é simples: se a expressão geradora ocupar mais de duas linhas, prefiro escrever uma função geradora por questões de legibilidade. Além do mais, como as funções geradoras têm nome, elas podem ser reutilizadas. Você sempre pode nomear uma expressão geradora e usá-la depois, atribuindo-a a uma variável, é claro, mas isso é ir longe demais com uma sintaxe que foi inventada para criar um gerador a ser usado somente no local onde foi declarado.

**Dica de sintaxe**

Quando uma expressão geradora é passada como o único argumento de uma função ou de um construtor, você não precisa codar um par de parênteses para a chamada da função e outro para a expressão geradora. Um único par será suficiente, como na chamada a `Vector` no método `__mul__` do exemplo 10.16, reproduzido aqui. Entretanto, se houver mais argumentos após a expressão geradora, será preciso colocá-la entre parênteses para evitar um `SyntaxError`:

```
def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented
```

Os exemplos que vimos com `Sentence` mostram o uso de geradores desempenhando o papel de iteradores clássicos: recuperando itens de uma coleção. Porém geradores também podem ser usados para produzir valores independentes de uma fonte de dados. A próxima seção mostra um exemplo desse caso.

Outro exemplo: gerador de progressão aritmética

O padrão `Iterator` clássico tem a ver com percorrer dados, ou seja, navegar por uma estrutura de dados. Mas uma interface-padrão baseada em um método para buscar o próximo item de uma série também é útil quando os itens são produzidos durante a execução, em vez de serem recuperados de uma coleção. Por exemplo, a função embutida `range` gera uma progressão aritmética (PA) de um intervalo de inteiros, e a função `itertools.count` gera uma PA ilimitada.

Discutiremos `itertools.count` na próxima seção, mas e se você precisar gerar uma PA contendo um intervalo de números de qualquer tipo?

O exemplo 14.10 mostra alguns testes de console para uma classe `ArithmeticProgression` que veremos em breve. A assinatura do construtor no exemplo 14.10 é `ArithmeticProgression(begin, step[, end])`. A função `range()` é semelhante a `ArithmeticProgression`, mas sua assinatura completa é `range(start, stop[, step])`. Decidi implementar uma assinatura diferente porque, em uma progressão aritmética, `step` é obrigatório, mas `end` é opcional. Também mudei os nomes dos argumentos de `start/stop` para `begin/end` para deixar bem claro que optei por uma assinatura diferente. Em cada teste do exemplo 14.10, chamo `list()` no resultado para inspecionar os valores gerados.

Exemplo 14.10 – Demonstração da classe `ArithmeticProgression`

```
>>> ap = ArithmeticProgression(0, 1, 3)
>>> list(ap)
[0, 1, 2]
>>> ap = ArithmeticProgression(1, .5, 3)
>>> list(ap)
[1.0, 1.5, 2.0, 2.5]
>>> ap = ArithmeticProgression(0, 1/3, 1)
>>> list(ap)
[0.0, 0.3333333333333333, 0.6666666666666666]
>>> from fractions import Fraction
>>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
>>> list(ap)
[Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
>>> from decimal import Decimal
>>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
>>> list(ap)
[Decimal('0.0'), Decimal('0.1'), Decimal('0.2')]
```

Observe que o tipo dos números na progressão aritmética resultante segue o tipo de `begin` ou de `step`, de acordo com as regras de coerção numérica da aritmética de Python. No exemplo 14.10, vemos listas de números `int`, `float`, `Fraction` e `Decimal`.

O exemplo 14.11 mostra a implementação da classe `ArithmeticProgression`.

Exemplo 14.11 – A classe `ArithmeticProgression`

```
class ArithmeticProgression:
```

```
    def __init__(self, begin, step, end=None): ❶
        self.begin = begin
        self.step = step
        self.end = end # None -> séries "infinitas"

    def __iter__(self):
        result = type(self.begin + self.step)(self.begin) ❷
        forever = self.end is None ❸
        index = 0
        while forever or result < self.end: ❹
            yield result ❺
            index += 1
            result = self.begin + self.step * index ❻
```

- ❶ `__init__` exige dois argumentos – `begin` e `step`. `end` é opcional. Se for `None`, a série será ilimitada.

- ❷ Essa linha produz um valor de `result` igual a `self.begin`, mas com coerção para o tipo das somas subsequentes.¹⁰
- ❸ Por questões de legibilidade, a flag `forever` será `True` se o atributo `self.end` for `None`, resultando em uma série ilimitada.
- ❹ Esse laço executa indefinidamente (`forever`) ou até que o resultado seja maior ou igual a `self.end`. Quando esse laço termina, o mesmo ocorre com a função.
- ❺ `result` atual é produzido.
- ❻ O próximo resultado em potencial é calculado. Talvez não seja entregue nunca, pois o laço `while` poderá terminar.

Na última linha do exemplo 14.11, em vez de simplesmente incrementar `result` com `self.step` iterativamente, optei pelo uso de uma variável `index` e calculei cada `result` somando `self.begin` a `self.step` multiplicado por `index` para reduzir o efeito cumulativo de erros ao trabalhar com números de ponto flutuante.

A classe `ArithmeticProgression` do exemplo 14.11 funciona conforme desejado, e é um exemplo claro de uso de uma função geradora para implementar o método especial `__iter__`. No entanto, se o objetivo principal de uma classe é criar um gerador implementando `__iter__`, a classe pode ser reduzida a uma função geradora. Uma função geradora, afinal de contas, é uma fábrica de geradores.

O exemplo 14.12 mostra uma função geradora chamada `aritprog_gen` que faz o mesmo trabalho de `ArithmeticProgression`, porém com menos código. Todos os testes do exemplo 14.10 passarão se você chamar `aritprog_gen` no lugar de `ArithmeticProgression`.¹¹

Exemplo 14.12 – A função geradora `aritprog_gen`

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

¹⁰ Em Python 2, havia uma função embutida `coerce()`, mas não existe mais em Python 3; foi considerada desnecessária porque as regras de coerção numérica estão implícitas nos métodos de operadores aritméticos. Desse modo, a melhor maneira que pude imaginar para fazer a coerção do valor inicial para o mesmo tipo do restante da série foi realizar a soma e usar seu tipo para converter o resultado. Fiz uma pergunta sobre isso na *Python-list* e obtive uma resposta excelente de Steven D'Aprano (<http://bit.ly/1J1b1YO>).

¹¹ O diretório *14-it-generator/* no repositório de código de *Python fluente* (<http://bit.ly/1J1tSti>) inclui doctests e um script `aritprog_runner.py`, que executa os testes para todas as variações dos scripts `aritprog*.py`.

O exemplo 14.12 é interessante, mas lembre-se sempre que há muitos geradores prontos para uso na biblioteca-padrão; na próxima seção, mostraremos uma implementação mais pythônica usando o módulo `itertools`.

Progressão aritmética com `itertools`

O módulo `itertools` em Python 3.4 tem 19 funções geradoras que podem ser combinadas de várias maneiras úteis.

Por exemplo, a função `itertools.count` devolve um gerador que produz números. Sem argumentos, ela produz uma série de inteiros começando em 0. Mas você pode fornecer os valores `start` e `step` opcionais para conseguir um resultado muito semelhante ao de nossas funções `aritprog_gen`:

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1
>>> next(gen)
1.5
>>> next(gen)
2.0
>>> next(gen)
2.5
```

No entanto `itertools.count` nunca para, portanto, se você chamar `list(count())`, Python tentará criar uma `list` maior que a memória disponível, e seu computador vai se comportar muito mal, bem antes de a chamada produzir uma exceção.

Por outro lado, temos a função `itertools.takewhile`; ela produz um gerador que consome outro gerador e para quando um dado predicado é avaliado como `False`. Desse modo, podemos combinar os dois e escrever:

```
>>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

Aproveitando `takewhile` e `count`, o exemplo 14.13 fica simples e compacto.

Exemplo 14.13 – `aritprog_v3.py`: funciona como as funções `aritprog_gen` anteriores

```
import itertools
```

```
def aritprog_gen(begin, step, end=None):
    first = type(begin + step)(begin)
```

```
ap_gen = itertools.count(first, step)
if end is not None:
    ap_gen = itertools.takewhile(lambda n: n < end, ap_gen)
return ap_gen
```

Observe que `aritprog_gen` não é uma função geradora no exemplo 14.13; ela não tem `yield` em seu corpo. Porém ela devolve um gerador, portanto funciona como uma fábrica de geradores, como faz uma função geradora.

O ponto principal do exemplo 14.13 é o seguinte: ao implementar geradores, confira o que está disponível na biblioteca-padrão; caso contrário, há uma boa chance de você reinventar a roda. É por isso que a próxima seção aborda várias funções geradoras prontas para uso.

Funções geradoras na biblioteca-padrão

A biblioteca-padrão oferece muitos geradores, desde objetos-arquivo de texto puro, que oferecem iteração linha a linha, até a fantástica função `os.walk` (<http://bit.ly/1HGqqwh>), que produz nomes de arquivos enquanto percorre uma árvore de diretório, deixando as buscas recursivas no sistema de arquivos tão simples quanto um laço `for`.

A função geradora `os.walk` é impressionante, mas, nesta seção, quero enfatizar as funções de propósito geral que aceitam iteráveis quaisquer como argumentos e devolvem geradores que produzem itens selecionados, calculados ou reorganizados. Resumi duas dúzias delas nas tabelas a seguir, desde funções embutidas até aquelas incluídas nos módulos `itertools` e `functools`. Por conveniência, agrupei-as por funcionalidade de alto nível, independentemente do local em que estão definidas.



Talvez você conheça todas as funções mencionadas nesta seção, mas algumas delas são subutilizadas, portanto uma visão geral rápida pode ser boa para lembrar o que já está disponível.

O primeiro grupo contém funções geradoras de filtragem: entregam um subconjunto de itens produzidos pelo iterável de entrada sem mudar os itens propriamente ditos. Usamos `itertools.takewhile` antes, neste capítulo, na seção “Progressão aritmética com `itertools`” na página 473. Como `takewhile`, a maioria das funções listadas na tabela 14.1 aceita um predicado, que é uma função booleana de um só argumento, aplicada a cada item da entrada para determinar se o item será produzido na saída.

Tabela 14.1 – Funções geradoras de filtragem

Módulo	Função	Descrição
itertools	compress(it, selector_it)	Consome dois iteráveis em paralelo; entrega itens de it sempre que o item correspondente em selector_it for verdadeiro
itertools	dropwhile(predicate, it)	Consome it descartando itens enquanto predicate for avaliado como verdadeiro; em seguida, produz todos os itens restantes (novas verificações não são feitas)
(função embutida)	filter(predicate, it)	Aplica predicate a cada item de it, entregando o item se predicate(item) for verdadeiro; se predicate for None, somente itens verdadeiros serão produzidos
itertools	filterfalse(predicate, it)	É o mesmo que filter, com a lógica de predicate negada; produz itens sempre que predicate for avaliada como falso
itertools	islice(it, stop) ou islice(it, start, stop, step=1)	Produz itens de uma fatia de it, semelhante a s[:stop] ou s[start:stop:step], exceto que it pode ser qualquer iterável e a operação é lazy
itertools	takewhile(predicate, it)	Produz itens enquanto predicate for avaliado como verdadeiro; em seguida para de produzir, e nenhuma verificação adicional é feita.

Alistagem de console no exemplo 14.14 mostra o uso de todas as funções da tabela 14.1.

Exemplo 14.14 – Exemplos de funções geradoras de filtragem

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1,0,1,1,0,1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
```

```
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

No próximo grupo estão os geradores de mapeamento: produzem itens computados a partir de cada item individual do iterável de entrada – ou iteráveis no caso de `map` e de `starmap`.¹² Os geradores da tabela 14.2 produzem um resultado por item dos iteráveis de entrada. Se a entrada for proveniente de mais de um iterável, a saída não será mais gerada assim que o primeiro iterável de entrada estiver esgotado.

Tabela 14.2 – Funções geradoras de mapeamento

Módulo	Função	Descrição
itertools	<code>accumulate(it, [func])</code>	Produz somas acumuladas; se <code>func</code> for especificada, entrega o resultado de sua aplicação ao primeiro par de itens; em seguida, ao primeiro resultado e o próximo item etc.
(função embutida)	<code>enumerate(iterable, start=0)</code>	Produz tuplas de dois elementos na forma <code>(index, item)</code> , em que <code>index</code> é contado a partir de <code>start</code> e <code>item</code> é obtido de <code>iterable</code>
(função embutida)	<code>map(func, it1, [it2, ..., itN])</code>	Aplica <code>func</code> a cada item de <code>it</code> , produzindo o resultado; se <code>N</code> iteráveis forem especificados, <code>func</code> deverá aceitar <code>N</code> argumentos e os iteráveis serão consumidos em paralelo
itertools	<code>starmap(func, it)</code>	Aplica <code>func</code> a cada item de <code>it</code> , produzindo o resultado; o iterável de entrada deve produzir itens <code>i</code> it iteráveis e <code>func</code> é aplicada como <code>func(*iit)</code>

O exemplo 14.15 mostra alguns usos de `itertools.accumulate`.

Exemplo 14.15 – Exemplos da função geradora `itertools.accumulate`

```
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> import itertools
>>> list(itertools.accumulate(sample)) # ❶
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min)) # ❷
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max)) # ❸
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul)) # ❹
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800] # ❺
```

¹² Nesse caso, o termo “mapeamento” não está relacionado a dicionários, mas à função embutida `map`.

- ❶ Soma cumulativa: cada item do resultado é o valor da soma até o item correspondente na entrada.
- ❷ Mínimo até o item correspondente na entrada.
- ❸ Máximo até o item correspondente na entrada.
- ❹ Produto cumulativo.
- ❺ Fatoriais de 1! a 10!.

As demais funções da tabela 14.2 estão no exemplo 14.16.

Exemplo 14.16 – Exemplos de funções geradoras de mapeamento

```
>>> list(enumerate('albatroz', 1)) # ❶
[(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7, 'o'), (8, 'z')]
>>> import operator
>>> list(map(operator.mul, range(11), range(11))) # ❷
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8])) # ❸
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8])) # ❹
[(0, 2), (1, 4), (2, 8)]
>>> import itertools
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1))) # ❺
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'oooooooo', 'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b/a,
... enumerate(itertools.accumulate(sample), 1))) # ❻
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
5.0, 4.375, 4.888888888888889, 4.5]
```

- ❶ Enumera as letras da palavra, começando em 1.
- ❷ Quadrados dos inteiros de 0 a 10.
- ❸ Multiplicando números de dois iteráveis em paralelo: resultado termina quando o iterável mais curto esgota.
- ❹ É isso que a função embutida `zip` faz.
- ❺ Repete cada letra da palavra de acordo com sua posição, começando em 1.
- ❻ Calcula a média.

Em seguida, temos o grupo de geradores de combinação – todos produzem itens a partir de vários iteráveis de entrada. `chain` e `chain.from_iterable` consomem os iteráveis

de entrada sequencialmente (um após o outro), enquanto `product`, `zip` e `zip_longest` consomem os iteráveis de entrada em paralelo. Veja a tabela 14.3.

Tabela 14.3 – Funções geradoras que combinam vários iteráveis de entrada

Módulo	Função	Descrição
itertools	<code>chain(it1, ..., itN)</code>	Produz todos os itens de <code>it1</code> , depois de <code>it2</code> etc., naturalmente
itertools	<code>chain.from_iterable(it)</code>	Produz todos os itens de cada iterável produzido por <code>it</code> , um após o outro, sem interrupções; <code>it</code> deve produzir uma série de iteráveis, por exemplo, uma lista de iteráveis
itertools	<code>product(it1, ..., itN, repeat=1)</code>	Produto cartesiano: produz tuplas de <code>N</code> elementos compostas da combinação de itens de cada iterável de entrada, como seriam produzidos por laços <code>for</code> aninhados; <code>repeat</code> permite que os iteráveis de entrada sejam consumidos mais de uma vez.
(função embutida)	<code>zip(it1, ..., itN)</code>	Produz tuplas de <code>N</code> elementos criadas a partir de itens obtidos dos iteráveis em paralelo, parando silenciosamente quando o primeiro iterável se esgotar.
itertools	<code>zip_longest(it1, ..., itN, fillvalue=None)</code>	Produz tuplas de <code>N</code> elementos criadas a partir de itens obtidos dos iteráveis em paralelo, parando somente quando o iterável mais longo esgotar, preenchendo as lacunas com <code>fillvalue</code>

O exemplo 14.17 mostra o uso das funções geradoras `itertools.chain` e `zip` e seus parentes próximos. Lembre-se de que a função `zip` deve seu nome ao zíper (não tem relação com compressão). Tanto `zip` quanto `itertools.zip_longest` foram apresentadas na seção “O fantástico zip” na página 336.

Exemplo 14.17 – Exemplos de funções geradoras de combinação

```
>>> list(itertools.chain('ABC', range(2))) # ❶
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC'))) # ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC'))) # ❸
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5))) # ❹
[('A', 0), ('B', 1), ('C', 2)]
>>> list(zip('ABC', range(5), [10, 20, 30, 40])) # ❺
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5))) # ❻
[('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
>>> list(itertools.zip_longest('ABC', range(5), fillvalue='?')) # ❼
[('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]
```

❶ `chain` normalmente é chamada com dois ou mais iteráveis.

- ❹ `chain` não tem nenhuma utilidade quando chamada com um único iterável.
- ❺ Mas `chain.from_iterable` pega cada item do iterável e os encadeia em sequência, desde que cada item seja, ele mesmo, um iterável.
- ❻ `zip` é comumente usada para combinar dois iteráveis em uma série de tuplas de dois elementos.
- ❼ Qualquer quantidade de iteráveis pode ser consumida por `zip` em paralelo, mas o gerador termina assim que o primeiro iterável esgota.
- ❽ `itertools.zip_longest` funciona como `zip`, porém ela consome todos os iteráveis de entrada até o fim, preenchendo as tuplas de saída com `None`, se for necessário.
- ❾ O argumento nomeado `fillvalue` especifica um valor personalizado para preenchimento.

O gerador `itertools.product` é um modo lazy de calcular produtos cartesianos, que implementamos usando list comprehensions com mais de uma cláusula `for` na seção “Produtos cartesianos” na página 49. Expressões geradoras com várias cláusulas `for` também podem ser usadas para gerar produtos cartesianos em modo lazy. O exemplo 14.18 mostra o uso de `itertools.product`.

Exemplo 14.18 – Exemplos da função geradora `itertools.product`

```
>>> list(itertools.product('ABC', range(2))) # ❶
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'spades hearts diamonds clubs'.split()
>>> list(itertools.product('AK', suits)) # ❷
[('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A', 'clubs'),
 ('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K', 'clubs')]
>>> list(itertools.product('ABC')) # ❸
[('A',), ('B',), ('C',)]
>>> list(itertools.product('ABC', repeat=2)) # ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
 ('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
>>> list(itertools.product(range(2), repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
 (1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> rows = itertools.product('AB', range(2), repeat=2)
>>> for row in rows: print(row)
...
('A', 0, 'A', 0)
('A', 0, 'A', 1)
('A', 0, 'B', 0)
('A', 0, 'B', 1)
```

```
('A', 1, 'A', 0)
('A', 1, 'A', 1)
('A', 1, 'B', 0)
('A', 1, 'B', 1)
('B', 0, 'A', 0)
('B', 0, 'A', 1)
('B', 0, 'B', 0)
('B', 0, 'B', 1)
('B', 1, 'A', 0)
('B', 1, 'A', 1)
('B', 1, 'B', 0)
('B', 1, 'B', 1)
```

- ❶ O produto cartesiano de uma str com três caracteres e um range com dois inteiros produz seis tuplas (porque 3 * 2 é 6).
- ❷ O produto de dois valores de carta de baralho ('AK') e quatro naipes é uma série de oito tuplas.
- ❸ Dado um único iterável, product produz uma série de tuplas de um elemento, o que não é muito útil.
- ❹ O argumento nomeado repeat=N informa product que ele deve consumir cada iterável de entrada N vezes.

Algumas funções geradoras expandem a entrada produzindo mais de um valor por item de entrada. Elas estão listadas na tabela 14.4.

Tabela 14.4 – Funções geradoras que expandem cada item de entrada em vários itens de saída

Módulo	Função	Descrição
itertools	combinations(it, out_len)	Produz combinações de out_len itens a partir dos itens entregues por it
itertools	combinations_with_replacement(it, out_len)	Produz combinações de out_len itens a partir dos itens entregues por it, incluindo combinações com itens repetidos
itertools	count(start=0, step=1)	Produz números começando em start, incrementados por step, indefinidamente
itertools	cycle(it)	Produz itens de it armazenando uma cópia de cada um e, em seguida, produz a sequência toda repetidamente e indefinidamente
itertools	permutations(it, out_len=None)	Produz permutações de out_len itens a partir dos itens produzidos por it; por padrão, out_len é len(list(it))
itertools	repeat(item, [times])	Produz o item dado repetidamente e indefinidamente, a menos que um número de vezes (times) seja especificado

As funções `count` e `repeat` de `itertools` devolvem geradores que produzem itens do nada: nenhuma delas aceita um iterável como entrada. Vimos `itertools.count` na seção “Progressão aritmética com `itertools`” na página 473. O gerador `cycle` faz um backup do iterável de entrada e entrega seus itens repetidamente. O exemplo 14.19 mostra o uso de `count`, `repeat` e `cycle`.

Exemplo 14.19 – `count`, `cycle` e `repeat`

```
>>> ct = itertools.count() # ❶
>>> next(ct) # ❷
0
>>> next(ct), next(ct), next(ct) # ❸
(1, 2, 3)
>>> list(itertools.islice(itertools.count(1, .3), 3)) # ❹
[1, 1.3, 1.6]
>>> cy = itertools.cycle('ABC') # ❺
>>> next(cy)
'A'
>>> list(itertools.islice(cy, 7)) # ❻
['B', 'C', 'A', 'B', 'C', 'A', 'B']
>>> rp = itertools.repeat(7) # ❼
>>> next(rp), next(rp)
(7, 7)
>>> list(itertools.repeat(8, 4)) # ❽
[8, 8, 8, 8]
>>> list(map(operator.mul, range(11), itertools.repeat(5))) # ❾
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

- ❶ Cria um gerador `count` e atribui a `ct`.
- ❷ Recupera o primeiro item de `ct`.
- ❸ Não posso criar uma `list` a partir de `ct`, pois `ct` não para nunca, portanto busco os três próximos itens.
- ❹ Posso criar uma `list` a partir de um gerador `count` se estiver limitado por `islice` ou `takewhile`.
- ❺ Cria um gerador `cycle` a partir de `'ABC'` e busca seu primeiro item, ou seja, `'A'`.
- ❻ Uma `list` só pode ser criada se estiver limitada por `islice`; os próximos sete itens são recuperados aqui.
- ❼ Cria um gerador `repeat` que produzirá o número 7 indefinidamente.
- ❽ Um gerador `repeat` pode ser limitado pelo argumento `times`: nesse caso, o número 8 será produzido 4 vezes.

- ⑨ Um uso comum de `repeat`: fornecer um argumento fixo em `map`; nesse caso, fornece o multiplicador 5.

As funções geradoras `combinations`, `combinations_with_replacement` e `permutations` – juntamente com `product` – são chamadas de *geradores combinatoriais* (combinatoric generators) na página de documentação de `itertools` (<http://bit.ly/py-itertools>). Também há uma relação próxima entre `itertools.product` e as demais funções *combinatórias*, como mostra o exemplo 14.20.

Exemplo 14.20 – Funções geradoras combinatoriais produzem diversos valores por item de entrada

```
>>> list(itertools.combinations('ABC', 2)) # ❶
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations_with_replacement('ABC', 2)) # ❷
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
>>> list(itertools.permutations('ABC', 2)) # ❸
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(itertools.product('ABC', repeat=2)) # ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
('C', 'A'), ('C', 'B'), ('C', 'C')]
```

- ❶ Todas as combinações de `len()==2` dos itens em 'ABC'; a ordem dos itens nas tuplas geradas é irrelevante (poderiam ser conjuntos).
- ❷ Todas as combinações de `len()==2` dos itens em 'ABC', incluindo combinações com itens repetidos.
- ❸ Todas as permutações de `len()==2` dos itens de 'ABC'; a ordem dos itens nas tuplas geradas é relevante.
- ❹ Produto cartesiano de 'ABC' e 'ABC' (esse é o efeito de `repeat=2`).

O último grupo de funções geradoras que discutiremos nesta seção foi projetado para entregar todos os itens dos iteráveis de entrada, mas reorganizados de alguma forma. Duas funções que devolvem múltiplos geradores são: `itertools.groupby` e `itertools.tee`. A outra função geradora desse grupo – a função embutida `reversed` – é a única discutida nesta seção que não aceita um iterável qualquer como entrada, mas apenas sequências. Isso faz sentido: como `reversed` produzirá itens do último para o primeiro, ele só funcionará com uma sequência de tamanho conhecido. Porém ele evita o custo de criar uma cópia invertida da sequência produzindo cada item conforme necessário. Coloquei a função `itertools.product` juntamente com os geradores de *combinação* na tabela 14.3 porque todas elas consomem mais de um iterável, enquanto todos os geradores da tabela 14.5 aceitam no máximo um iterável de entrada.

Tabela 14.5 – Funções geradoras de reorganização

Módulo	Função	Descrição
itertools	groupby(it, key=None)	Produz tuplas de dois elementos na forma (key, group), em que key é o critério de agrupamento e group é um gerador que produz itens no grupo.
(função embutida)	reversed(seq)	Produz itens de seq na ordem inversa, do último para o primeiro; seq deve ser uma sequência ou deve implementar o método especial <code>__reversed__</code>
itertools	tee(it, n=2)	Produz uma tupla de <i>n</i> geradores, cada um entregando os itens do iterável de entrada de modo independente

O exemplo 14.21 mostra o uso de `itertools.groupby` e da função embutida `reversed`. Note que `itertools.groupby` supõe que o iterável de entrada está ordenado pelo critério de agrupamento ou, no mínimo, que os itens estão juntos de acordo com esse critério – apesar de não estarem ordenados.

Exemplo 14.21 – `itertools.groupby`

```
>>> list(itertools.groupby('LLLLAAGGG')) # ❶
[('L', <itertools._grouper object at 0x102227cc0>),
 ('A', <itertools._grouper object at 0x102227b38>),
 ('G', <itertools._grouper object at 0x102227b70>)]
>>> for char, group in itertools.groupby('LLLLAAAGG'): # ❷
...     print(char, '->', list(group))
...
L -> ['L', 'L', 'L', 'L']
A -> ['A', 'A',]
G -> ['G', 'G', 'G']
>>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
...            'bat', 'dolphin', 'shark', 'lion']
>>> animals.sort(key=len) # ❸
>>> animals
['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
 'giraffe', 'dolphin']
>>> for length, group in itertools.groupby(animals, len): # ❹
...     print(length, '->', list(group))
...
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
>>> for length, group in itertools.groupby(reversed(animals), len): # ❺
```

```
...     print(length, '->', list(group))
...
7 -> ['dolphin', 'giraffe']
5 -> ['shark', 'eagle']
4 -> ['lion', 'bear', 'duck']
3 -> ['bat', 'rat']
>>>
```

- ❶ groupby produz tuplas de (key, group_generator).
- ❷ Tratar geradores groupby envolve iteração aninhada: nesse caso, o laço for externo e o construtor list interno.
- ❸ Para usar groupby, a entrada deve estar ordenada; nesse caso, as palavras estão ordenadas por tamanho.
- ❹ Novamente, percorre o par key e group em um laço para exibir key e expande group em uma list.
- ❺ Aqui o gerador reverse é usado para uma iteração por animals, da direita para a esquerda.

A última função geradora desse grupo é `iterator.tee`, que tem um comportamento curioso: produz vários geradores a partir de um único iterável de entrada, cada um produzindo todos os itens da entrada. Esses geradores podem ser consumidos de modo independente, como mostra o exemplo 14.22.

Exemplo 14.22 – `itertools.tee` produz vários geradores, cada um entregando todos os itens do gerador de entrada

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x10222abc8>, <itertools._tee object at 0x10222ac08>]
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
>>> next(g2)
'B'
>>> list(g1)
['B', 'C']
>>> list(g2)
['C']
>>> list(zip(*itertools.tee('ABC')))
[('A', 'A'), ('B', 'B'), ('C', 'C')]
```

Observe que vários exemplos desta seção usaram combinações de funções geradoras. Esse é um ótimo recurso dessas funções: como quase todas aceitam geradores como argumentos e devolvem geradores, elas podem ser combinadas de várias maneiras diferentes.

Por falar em combinação de geradores, o comando `yield from`, novo em Python 3.3, é uma ferramenta que serve exatamente para isso.

Nova sintaxe em Python 3.3: `yield from`

Laços `for` aninhados são a solução tradicional quando uma função geradora precisa entregar valores produzidos por outro gerador.

Por exemplo, veja uma implementação caseira de um gerador de encadeamento parecido com a função `itertools.chain`:¹³

```
>>> def chain(*iterables):
...     for it in iterables:
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

A função geradora `chain` delega a cada iterável recebido, um de cada vez. A *PEP 380 – Syntax for Delegating to a Subgenerator* (Sintaxe para delegar a um subgerador, <http://bit.ly/1wpQv0i>) introduziu uma nova sintaxe para isso, como mostra a próxima listagem de console:

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i
...
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

Como podemos ver, `yield from i` substitui totalmente o laço `for` interno. O uso de `yield from` nesse exemplo está correto, e o código é mais legível, mas parece ser apenas açúcar sintático. Acontece que, além de substituir um laço, `yield from` cria um canal que conecta o gerador interno diretamente ao cliente do gerador externo. Esse canal torna-se muito importante quando os geradores são usados como corrotinas e não

¹³ `itertools.chain` da biblioteca-padrão está implementada em C.

apenas produzem, mas também consomem valores do código do cliente. O capítulo 16 explora as corrotinas e tem várias páginas que explicam por que `yield from` é muito mais que açúcar sintático.

Depois desse primeiro contato com `yield from`, retornaremos à nossa análise das funções especializadas em iteráveis da biblioteca-padrão.

Funções de redução de iteráveis

Todas as funções na tabela 14.6 aceitam um iterável e devolvem um único resultado. São conhecidas como funções de “redução” (reducing), “folding” (dobra) ou “acumuladora” (accumulating). Na verdade, todas as funções embutidas listadas aqui podem ser implementadas com `functools.reduce`, mas existem como funções embutidas para facilitar alguns casos de uso comuns. Além disso, no caso de `all` e `any`, há uma otimização importante que não pode ser feita com `reduce`: essas funções são interrompidas (ou seja, param de consumir o iterador assim que o resultado é determinado). Veja o último teste com `any` no exemplo 14.23.

Tabela 14.6 – Funções embutidas que leem iteráveis e devolvem valores únicos

Módulo	Função	Descrição
(função embutida)	<code>all(it)</code>	Devolve <code>True</code> se todos os itens em <code>it</code> forem verdadeiros; caso contrário, devolve <code>False</code> ; <code>all([])</code> devolve <code>True</code>
(função embutida)	<code>any(it)</code>	Devolve <code>True</code> se algum item de <code>it</code> for verdadeiro; caso contrário, devolve <code>False</code> ; <code>any([])</code> devolve <code>False</code>
(função embutida)	<code>max(it, [key=,] [default=])</code>	Devolve o valor máximo dos itens em <code>it</code> ; ^a <code>key</code> é uma função de ordenação, como em <code>sorted</code> ; <code>default</code> é devolvido se o iterável estiver vazio
(função embutida)	<code>min(it, [key=,] [default=])</code>	Devolve o valor mínimo dos itens em <code>it</code> . ^b <code>key</code> é uma função de ordenação, como em <code>sorted</code> ; <code>default</code> é devolvido se o iterável estiver vazio
<code>functools</code>	<code>reduce(func, it, [initial])</code>	Devolve o resultado da aplicação de <code>func</code> ao primeiro par de itens, em seguida ao resultado e ao terceiro item, e assim sucessivamente; se fornecido, <code>initial</code> compõe o par inicial com o primeiro item
(função embutida)	<code>sum(it, start=0)</code>	A soma de todos os itens em <code>it</code> , com o valor <code>start</code> opcional somado (use <code>math.fsum</code> para uma precisão melhor quando somar números de ponto flutuante)

a Também pode ser chamado como `max(arg1, arg2, ..., [key=?])`, caso em que o argumento máximo é devolvido.

b Também pode ser chamado como `min(arg1, arg2, ..., [key=?])`, caso em que o argumento mínimo é devolvido.

O funcionamento de `all` e de `any` é mostrado no exemplo 14.23.

Exemplo 14.23 – Resultados de `all` e de `any` para algumas sequências

```
>>> all([1, 2, 3])
True
>>> all([1, 0, 3])
False
>>> all([])
True
>>> any([1, 2, 3])
True
>>> any([1, 0, 3])
True
>>> any([0, 0.0])
False
>>> any([])
False
>>> g = (n for n in [0, 0.0, 7, 8])
>>> any(g)
True
>>> next(g)
8
```

Uma explicação mais longa de `functools.reduce` está na seção “Vector tomada #4: hashing e um `==` mais rápido” na página 332.

Outra função embutida que aceita um iterável e devolve algo diferente é `sorted`. Diferente de `reversed`, que é uma função geradora, `sorted` cria e devolve uma lista. Afinal de contas, todos os itens do iterável de entrada precisam ser lidos para que possam ser ordenados, e a ordenação ocorre em uma `list`; sendo assim, `sorted` simplesmente devolve essa `list` depois de ter terminado. Mencionei `sorted` aqui porque ele consome um iterável qualquer.

É claro que `sorted` e as funções de redução funcionam somente com iteráveis que param em algum momento. Caso contrário, elas continuarão coletando itens e jamais devolverão um resultado.

Vamos agora voltar para a função embutida `iter()`: ela tem um recurso pouco conhecido que ainda não discutimos.

Uma visão mais detalhada da função `iter`

Como vimos, Python chama `iter(x)` quando precisa fazer uma iteração por um objeto `x`.

No entanto `iter` tem outro truque: pode ser chamada com dois argumentos para criar um iterador a partir de uma função normal ou qualquer objeto invocável. Nesse uso, o primeiro argumento deve ser um invocável a ser chamado repetidamente (sem argumentos) para produzir valores e o segundo argumento é uma sentinela – um marcador que, quando devolvido pelo invocável, faz o iterador levantar `StopIteration` em vez de entregar a sentinela.

O exemplo a seguir mostra como usar `iter` para lançar um dado de seis lados até que 1 seja tirado:

```
>>> def d6():
...     return randint(1, 6)
...
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x00000000029BE6A0>
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

Observe que a função `iter`, nesse caso, devolve um `callable_iterator`. O laço `for` do exemplo pode executar por muito tempo, mas jamais exibirá 1, pois esse é o valor da sentinela. Como ocorre normalmente com iteradores, o objeto `d6_iter` do exemplo perde sua utilidade quando esgota. Para reiniciar, você deve recriar o iterador chamando `iter(_)` novamente.

Um exemplo útil está na documentação da função embutida `iter` (<http://bit.ly/1HGqw70>). O trecho de código a seguir lê linhas de um arquivo até que uma linha em branco seja encontrada ou o final do arquivo seja alcançado:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

Para encerrar este capítulo, apresentarei um exemplo prático do uso de geradores para tratar um grande volume de dados de modo eficiente.

Estudo de caso: geradores em um utilitário para conversão de banco de dados

Alguns anos atrás, trabalhei na BIREME: uma biblioteca digital administrada pela OPAS/OMS (Organização Pan-Americana da Saúde/Organização Mundial da Saúde) em São Paulo. Entre os conjuntos de dados bibliográficos criados pela BIREME estão a LILACS (Literatura da América Latina e Caribe em Ciências da Saúde) e o SciELO (Scientific Electronic Library Online), dois bancos de dados abrangentes que indexam a literatura científica e técnica produzida na região.

Desde o final dos anos 80, o sistema de banco de dados usado para administrar a LILACS é o CDS/ISIS, um banco de dados documental, não relacional, criado pela UNESCO e reescrito em C pela BIREME para rodar em servidores GNU/Linux. Uma de minhas tarefas era pesquisar alternativas para uma possível migração da LILACS – e em algum momento, do SciELO, uma base muito maior – para um banco de dados documental moderno e de código aberto como o CouchDB ou o MongoDB.

Como parte dessa pesquisa, escrevi um script Python, *isis2json.py*, que lê um arquivo de CDS/ISIS e escreve um arquivo JSON adequado para importar para CouchDB ou MongoDB. Inicialmente, o script lia arquivos no formato ISO-2709 exportado pelo CDS/ISIS. A leitura e a escrita tiveram de ser feitas de forma incremental, pois os conjuntos de dados completos eram muito maiores que a memória RAM disponível. Foi bem fácil: cada iteração do laço `for` principal lia um registro do arquivo *.iso*, processava-o e gravava na saída *.json*.

Entretanto, por motivos operacionais, foi considerado necessário que *isis2json.py* aceitasse outro formato de dados de CDS/ISIS: os arquivos binários *.mst* usados em produção na BIREME – para evitar a custosa exportação para ISO-2709.

Agora eu tinha um novo problema: a biblioteca para ler ISO-2709 e a biblioteca para ler arquivos *.mst* tinham APIs bem diferentes. Além disso, o laço que escrevia JSON já era complicado porque o script aceitava várias opções de linha de comando para reestruturar cada registro de saída. Ler dados usando duas APIs diferentes no mesmo laço `for` em que o JSON era produzido seria impraticável.

A solução foi isolar a lógica de leitura em um par de funções geradoras: uma para cada formato de entrada aceito. No final, o script *isis2json.py* foi dividido em quatro funções. Você pode ver o script principal em Python 2 no exemplo A-5, mas o código-fonte completo, com as dependências, está em *fluentpython/isis2json* (<http://bit.ly/1HGqzzT>) no GitHub.

Veja como o script está estruturado de modo geral:

main

A função `main` usa `argparse` para ler opções de linha de comando que configuram a estrutura dos registros de saída. De acordo com a extensão do arquivo de entrada, uma função geradora adequada é selecionada para ler os dados e produzir os registros, um a um.

`iter_iso_records`

Essa função geradora lê arquivos `.iso` (supondo que estejam no formato ISO-2709). Ela aceita dois argumentos: o nome do arquivo e `isis_json_type`, que é uma das opções relacionadas à estrutura do registro. Cada iteração de seu laço `for` lê um registro, cria um `dict` vazio, preenche-o com dados nos campos e produz o `dict`.

`iter_mst_records`

Essa outra função geradora lê arquivos `.mst`.¹⁴ Se observar o código-fonte de `isis2json.py`, verá que ele não é tão simples quanto `iter_iso_records`, mas sua interface e a estrutura geral são as mesmas: ele aceita um nome de arquivo e um argumento `isis_json_type` e entra em um laço `for`, que cria e produz um `dict` por iteração, representando um único registro.

`write_json`

Essa função faz a escrita dos registros JSON, um de cada vez. Ela aceita vários argumentos, mas o primeiro – `input_gen` – é uma referência a uma função geradora: `iter_iso_records` ou `iter_mst_records`. O laço `for` principal em `write_json` itera pelos dicionários produzidos pelo gerador `input_gen` selecionado, processa-o de diversas maneiras conforme determinado pelas opções de linha de comando e escreve o registro JSON no final do arquivo de saída.

Aproveitando as funções geradoras, consegui desacoplar a lógica de leitura da lógica de escrita. É claro que a maneira mais simples de desacoplá-las seria ler todos os registros na memória e, em seguida, gravá-los em disco. Porém essa não era uma opção viável por causa do tamanho das massas de dados. Ao usar geradores, a leitura e a escrita ficaram sincronizadas, de modo que o script pode processar arquivos de qualquer tamanho.

Se `isis2json.py` precisar aceitar um formato de entrada adicional – por exemplo, MARCXML, um DTD usado pela Biblioteca do Congresso dos Estados Unidos para representar dados ISO-2709 –, será fácil acrescentar uma terceira função geradora para implementar a lógica de leitura sem mudar nada na complicada função `write_json`.

¹⁴ Na verdade, a biblioteca usada para ler o complicado formato binário `.mst` está escrita em Java, portanto essa funcionalidade está disponível somente quando `isis2json.py` é executado com o interpretador Jython, versão 2.5 ou mais recente. Para mais detalhes, consulte o arquivo `README.rst` (<http://bit.ly/1MM5aXD>) no repositório. As dependências são importadas nas funções geradoras que precisam delas, portanto o script pode ser executado mesmo quando somente uma das bibliotecas externas estiver disponível.

Não é nenhuma engenharia espacial, mas é um exemplo real em que geradores ofereceram uma solução flexível para processar conjuntos de dados como um stream de registros, mantendo baixo o uso de memória, independentemente do volume de dados. Qualquer pessoa que administre massas de dados grandes terá muitas oportunidades de usar geradores na prática.

A próxima seção menciona um aspecto dos geradores do qual, por enquanto, vamos passar por cima. Continue lendo para entender por quê.

Geradores como corrotinas

Aproximadamente cinco anos depois da introdução das funções geradoras com a palavra reservada `yield` em Python 2.2, a PEP 342 – *Coroutines via Enhanced Generators* (Corrotinas com geradores melhorados, <https://www.python.org/dev/peps/pep-0342/>) foi implementada em Python 2.5. Essa proposta resultou na adição de métodos e funcionalidades extras aos objetos geradores, com destaque para o método `.send()`.

Como `__next__()`, `.send()` faz o gerador avançar para o próximo `yield`, mas também permite que o cliente que está usando o gerador envie dados a ele; qualquer argumento passado para `.send()` será o valor da expressão `yield` correspondente no corpo da função geradora. Em outras palavras, `.send()` permite uma troca de dados bidirecional entre o código do cliente e o gerador – em comparação com `__next__()`, que só permite que o cliente receba dados do gerador.

Essa é uma “melhoria” tão importante que, na verdade, altera a natureza dos geradores; quando usados dessa maneira, eles se tornam *corrotinas*. David Beazley – provavelmente o escritor e palestrante mais empenhado em estudar corrotinas na comunidade Python – passou as seguintes mensagens em um famoso tutorial na PyCon US 2009 (<http://www.dabeaz.com/coroutines/>):

- geradores produzem dados para iteração;
- corrotinas são consumidoras de dados;
- para evitar que seu cérebro exploda, não misture os dois conceitos;
- corrotinas não estão relacionadas com iteração;
- Observação: há uma utilidade em fazer `yield` produzir um valor em uma corrotina, mas não está relacionada à iteração.¹⁵

— David Beazley

“A Curious Course on Coroutines and Concurrency”

¹⁵ Slide 33, “Keeping It Straight” (Esclarecendo) em “A Curious Course on Coroutines and Concurrency” (Um curioso curso sobre corrotinas e concorrência, <http://www.dabeaz.com/coroutines/Coroutines.pdf>).

Seguirei o conselho de Dave e encerrarei este capítulo – cujo assunto são técnicas de iteração – sem falar de `send` e de outros recursos que permitem usar geradores como corrotinas. As corrotinas serão discutidas no capítulo 16.

Resumo do capítulo

A iteração está tão integrada na linguagem que gosto de dizer que Python “saca” (groks) os iteradores.¹⁶ A integração do padrão Iterator na semântica de Python é um excelente exemplo de como os padrões de projeto não são igualmente aplicáveis em todas as linguagens de programação. Em Python, um iterador clássico implementado “na unha” como no exemplo 14.4 não tem uso prático, exceto como exemplo didático.

Neste capítulo, criamos algumas versões de uma classe para iterar por cada palavra de um arquivo-texto que podem ser bem grandes. Graças ao uso de geradores, as sucessivas refatorações da classe `Sentence` ficaram menores e mais legíveis – quando você sabe como elas funcionam.

Em seguida, codamos um gerador de progressões aritméticas e mostramos como aproveitar o módulo `itertools` para simplificá-lo. Uma visão geral das 24 funções geradoras de propósito geral na biblioteca-padrão foi apresentada a seguir.

Depois disso, vimos a função embutida `iter`: em primeiro lugar, para ver como ela devolve um iterador quando chamada como `iter(o)` e, em seguida, estudamos como ela cria um iterador a partir de qualquer função quando chamada como `iter(func, sentinel)`.

Para ter um contexto prático, descrevi a implementação de um utilitário para conversão de banco de dados usando funções geradoras para desacoplar a lógica da leitura da lógica de escrita, permitindo um tratamento eficiente de massas de dados grandes e facilitando o suporte a mais de um formato de entrada de dados.

Também mencionei neste capítulo a sintaxe `yield from`, nova em Python 3.3, e as corrotinas. Esses dois assuntos foram apenas introduzidos aqui e serão detalhados mais adiante no livro.

¹⁶ De acordo com o *Jargon file* (<http://catb.org/~esr/jargon/html/G/grok.html>), o verbo *grok* não significa simplesmente aprender algo, mas absorvê-lo a ponto de se “tornar parte de você mesmo, parte de sua identidade”.

Leituras complementares

Uma explicação técnica detalhada sobre os geradores está em *The Python Language Reference* (Guia de referência à linguagem Python) na seção 6.2.9. *Yield expressions* (Expressões com yield, <http://bit.ly/1MM5Xb5>). A PEP em que as funções geradoras foram definidas é a PEP 255 — *Simple Generators* (Geradores simples, <https://www.python.org/dev/peps/pep-0255/>).

A documentação do módulo `itertools` (<https://docs.python.org/3/library/itertools.html>) é excelente por causa de todos os exemplos incluídos. Embora as funções nesse módulo estejam implementadas em C, a documentação mostra como várias delas seriam implementadas em Python, muitas vezes aproveitando outras funções do módulo. Os exemplos de uso são ótimos: por exemplo, há um trecho de código que mostra como usar a função `accumulate` para amortizar um empréstimo com juros, dada uma lista de pagamentos ao longo do tempo. Há também uma seção *Itertools Recipes* (Receitas com itertools, <http://bit.ly/1MM5YvA>), contendo funções adicionais de alto desempenho que usam funções de `itertools` como blocos de construção.

O capítulo 4, “Iterators and Generators” (Iteradores e geradores) do livro *Python Cookbook*, 3E (O’Reilly)¹⁷, de David Beazley e Brian K. Jones, tem 16 receitas que abordam esse assunto de vários ângulos diferentes, sempre com enfoque em aplicações práticas.

A sintaxe `yield from` é explicada com exemplos em *What’s New in Python 3.3* [O que há de novo em Python 3.3; veja a PEP 380: *Syntax for Delegating to a Subgenerator* (Sintaxe para delegar a um subgerador, <http://bit.ly/1MM6d9R>)]. Também discutiremos essa sintaxe em detalhes nas seções “Usando yield from” na página 530 e “O significado de yield from” na página 536 no capítulo 16.

Se estiver interessado em bancos de dados documentais e quiser saber mais sobre o contexto de “Estudo de caso: geradores em um utilitário para conversão de banco de dados” na página 489, o Code4Lib Journal – que aborda a intersecção entre bibliotecas e tecnologia – publicou meu artigo “From ISIS to CouchDB: Databases and Data Models for Bibliographic Records” (De ISIS ao CouchDB: bancos de dados e modelos de dados para registros bibliográficos, <http://journal.code4lib.org/articles/4893>). Uma das seções do artigo descreve o script `isis2json.py`. O restante explica por que e como o modelo de dados semiestruturado implementado por bancos de dados de documentais como CouchDB e MongoDB são mais adequados que o modelo relacional para o gerenciamento colaborativo de coleções de dados bibliográficos.

¹⁷ Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

Ponto de vista

Sintaxe de funções geradoras: mais açúcar seria bom

Os designers devem garantir que controles e mostradores para propósitos diferentes são significativamente diferentes uns dos outros.

— Donald Norman

The Design of Everyday Things

O código-fonte desempenha o papel de “controles e mostradores” em linguagens de programação. Acho que Python tem um design excepcional; seu código-fonte, com frequência, é tão legível quanto um pseudocódigo. Mas nada é perfeito. Guido van Rossum deveria ter seguido o conselho de Donald Norman (autor da citação anterior) e introduzido outra palavra reservada para definir expressões geradoras, em vez de reutilizar `def`. A seção “BDFL Pronouncements” (Pronunciamentos do BDFL) da PEP 255 — *Simple Generators* (Geradores simples, <https://www.python.org/dev/peps/pep-0255/>), de fato, menciona este argumento:

Um comando “yield” incluído no corpo não é suficiente para avisar que a semântica é tão diferente.

Porém Guido odeia introduzir novas palavras reservadas e não achou esse argumento convincente, por isso acabamos ficando com `def`.

Reutilizar a sintaxe de funções em geradores tem outras consequências ruins. No artigo e trabalho experimental “Python, the Full Monty: A Tested Semantics for the Python Programming Language” (Python completo: uma semântica testada para a linguagem de programação Python), Politz¹⁸ et al. mostram o seguinte exemplo trivial de uma função geradora (Seção 4.1 do artigo):

```
def f(): x=0
    while True:
        x += 1
        yield x
```

Os autores então destacam que não podemos abstrair o processo de produção (yielding) com uma chamada de função (Exemplo 14.24).

¹⁸ Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu e Shriram Krishnamurthi, “Python: The Full Monty”, SIGPLAN Not. 48, 10 (outubro de 2013), 217-232.

Exemplo 14.24 – “[Isso] parece realizar uma abstração simples sobre o processo de produção (yielding)” (Politz et al.)

```
def f():  
    def do_yield(n):  
        yield n  
    x = 0  
    while True:  
        x += 1  
        do_yield(x)
```

Se chamarmos `f()` no exemplo 14.24, teremos um laço infinito, e não um gerador, pois a palavra reservada `yield` faz apenas a função imediatamente ao redor dela ser uma função geradora. Embora as funções geradoras se pareçam com funções, não podemos delegar a outra função geradora com uma simples chamada de função. Para comparar, a linguagem Lua não impõe essa limitação. Uma corrotina em Lua pode chamar outras funções, e qualquer uma pode entregar valores a quem fez originalmente a chamada.

A nova sintaxe `yield from` foi introduzida para permitir que um gerador ou uma corrotina em Python delegasse o trabalho a outro, sem exigir a solução alternativa de um laço `for` interno. O exemplo 14.24 pode ser “corrigido” prefixando a chamada da função com `yield from`, como mostra o exemplo 14.25.

Exemplo 14.25 – Esse código faz uma abstração simples no processo de produção de valores

```
def f():  
    def do_yield(n):  
        yield n  
    x = 0  
    while True:  
        x += 1  
        yield from do_yield(x)
```

Reutilizar `def` para declarar geradores foi um erro de usabilidade, e o problema piorou em Python 2.5 com as corrotinas, que também são implementadas como funções com `yield`. No caso de corrotinas, `yield`, por acaso, aparece – normalmente – do lado direito de uma atribuição porque recebe o argumento da chamada a `.send()` do cliente. Como diz David Beazley:

Apesar de algumas semelhanças, geradores e corrotinas são basicamente dois conceitos diferentes.¹⁹

¹⁹ Slide 31, “A Curious Course on Coroutines and Concurrency” (Um curioso curso sobre corrotinas e concorrência, <http://www.dabeaz.com/coroutines/Coroutines.pdf>).

Acredito que as corrotinas também mereciam sua própria palavra reservada. Como veremos depois, as corrotinas frequentemente são usadas com decoradores especiais, que as diferenciam de outras funções. Porém as funções geradoras não são decoradas com tanta frequência, de modo que precisamos percorrer seus corpos em busca de `yield` para perceber que elas não são funções, mas uma criatura totalmente diferente.

Pode-se argumentar que, pelo fato de esses recursos terem sido criados para funcionar com pouca sintaxe adicional, uma sintaxe extra seria apenas “açúcar sintático”. Acontece que gosto de açúcar sintático quando isso faz recursos diferentes parecerem diferentes. A falta de açúcar sintático é o principal motivo pelo qual códigos em Lisp são difíceis de ler: todas as construções da linguagem Lisp se parecem com uma chamada de função.

Semântica de gerador versus iterador

Há pelo menos três maneiras de pensar no relacionamento entre iteradores e geradores. A primeira é do ponto de vista da interface. O protocolo de iteradores de Python define dois métodos: `__next__` e `__iter__`. Objetos geradores implementam ambos, portanto, desse ponto de vista, todo gerador é um iterador. De acordo com essa definição, objetos criados pela função embutida `enumerate()` são iteradores:

```
>>> from collections import abc
>>> e = enumerate('ABC')
>>> isinstance(e, abc.Iterator)
True
```

A segunda é do ponto de vista da implementação. Desse ângulo, um gerador é uma construção da linguagem Python que pode ser implementada de duas maneiras: como uma função com a palavra reservada `yield` ou como uma expressão geradora. Os objetos geradores resultantes da chamada a uma função geradora ou da avaliação de uma expressão geradora são instâncias de um tipo interno chamado `GeneratorType` (<http://bit.ly/1MM6Sbm>). Desse ponto de vista, todo gerador também é um iterador, pois as instâncias de `GeneratorType` implementam a interface de iteradores. Mas você pode escrever um iterador que não seja um gerador – implementando o padrão `Iterator` clássico, como vimos no exemplo 14.4, ou codando uma extensão em C. Os objetos `enumerate` não são geradores desse ponto de vista:

```
>>> import types
>>> e = enumerate('ABC')
>>> isinstance(e, types.GeneratorType)
False
```

Isso acontece porque `types.GeneratorType` (<https://docs.python.org/3/library/types.html#types.GeneratorType>) é definido como “o tipo de objeto gerador-iterador produzido pela chamada a uma função geradora”.

A terceira maneira de pensar no relacionamento entre iteradores e geradores é do ponto de vista conceitual. No padrão de projeto *Iterator* clássico – conforme definido no livro da *GoF* –, o iterador percorre uma coleção e entrega seus itens. O iterador pode ser bem complexo; por exemplo, pode navegar por uma estrutura de dados do tipo árvore. Porém, independentemente da lógica que estiver em um iterador clássico, ele sempre lerá valores de uma fonte de dados existente e, quando você chamar `next(it)`, não se espera que o iterador altere o item da fonte; espera-se que ele simplesmente entregue-o como está.

Em comparação, um gerador pode produzir valores sem necessariamente percorrer uma coleção, como faz `range`. Mesmo estando associados a uma coleção, os geradores não estão limitados a entregar somente os itens contidos nela, mas podem entregar outros valores derivados deles. Um exemplo claro disso é a função `enumerate`. De acordo com a definição original do padrão de projeto, o gerador devolvido por `enumerate` não é um iterador, pois ele cria as tuplas que produz.

Nesse nível conceitual, a técnica de implementação é irrelevante. Você pode codar um gerador sem usar um objeto gerador em Python. O exemplo 14.26 é um gerador de Fibonacci que escrevi somente para enfatizar essa questão.

Exemplo 14.26 – `fibonacci_by_hand.py`: gerador de Fibonacci sem instâncias de `GeneratorType`

```
class Fibonacci:
```

```
    def __iter__(self):
        return FibonacciGenerator()
```

```
class FibonacciGenerator:
```

```
    def __init__(self):
        self.a = 0
        self.b = 1

    def __next__(self):
        result = self.a
        self.a, self.b = self.b, self.a + self.b
        return result

    def __iter__(self):
        return self
```

O exemplo 14.26 funciona, mas é apenas um exemplo tolo. Veja o gerador de Fibonacci pythônico:

```
def fibonacci():  
    a, b = 0, 1  
    while True:  
        yield a  
        a, b = b, a + b
```

É claro que você sempre pode usar a sintaxe de geradores para executar as tarefas básicas de um iterador: percorrer uma coleção e entregar seus itens.

Na verdade, programadores Python não são rígidos quanto a essa distinção: geradores também são chamados de iteradores, mesmo nas documentações oficiais. A definição canônica de um iterador no Glossário de Python (<https://docs.python.org/dev/glossary.html#term-iterator>) é tão genérica que inclui tanto iteradores quanto geradores:

Iterador: um objeto que representa um stream de dados. [...]

Vale a pena ler a definição completa de *iterador* (<https://docs.python.org/3/glossary.html#term-iterator>) no Glossário de Python. Por outro lado, a definição de *gerador* (<https://docs.python.org/3/glossary.html#term-generator>) nesse glossário trata *iterador* e *gerador* como sinônimos e usa a palavra “gerador” para referir-se tanto à função geradora quanto ao objeto gerador que ela cria. Portanto, no jargão da comunidade Python, iterador e gerador são sinônimos bem próximos.

A interface minimalista de iteradores em Python

Na seção “Implementation” (Implementação) do padrão Iterator²⁰, a *Gangue dos Quatro* escreveu o seguinte:

A interface mínima de Iterator é constituída das operações First, Next, IsDone e CurrentItem.

No entanto essa mesma frase tem uma nota de rodapé em que se lê:

Podemos deixar essa interface menor ainda combinando Next, IsDone e CurrentItem em uma única operação que avança para o próximo objeto e o devolve. Se o percurso terminar, essa operação devolverá um valor especial (por exemplo, 0) que marca o fim da iteração.

É parecido com o que temos em Python: o método único `__next__` faz o trabalho. Porém, em vez de usar uma sentinela, que poderia ser ignorada por engano, a exceção `StopIteration` sinaliza o fim da iteração. Simples e correto. Esse é o jeito Python.

CAPÍTULO 15

Gerenciadores de contexto e blocos else

Gerenciadores de contexto poderão se tornar tão importantes quanto a própria sub-rotina. Só os exploramos superficialmente. [...] Basic tem uma instrução `with`; há instruções `with` em muitas linguagens. Mas elas não fazem a mesma coisa; todas executam uma tarefa muito rasa, evitam que você precise fazer buscas repetidas [de atributos] usando ponto, não fazem nem desfazem configurações. Não pense que são iguais só porque têm o mesmo nome. A instrução `with` faz muito mais.¹

— Raymond Hettinger
Evangelista eloquente de Python

Neste capítulo, discutiremos recursos de controle de fluxo não muito comuns em outras linguagens que, por esse motivo, tendem a ser ignorados ou subutilizados em Python. São eles:

- O comando `with` e os gerenciadores de contexto (context managers)
- A cláusula `else` em comandos `for`, `while` e `try`

O comando `with` define um contexto temporário e o desfaz de modo confiável, sob controle de um objeto gerenciador de contexto. Isso evita erros e reduz códigos repetitivos, deixando as APIs, ao mesmo tempo, mais seguras e mais fáceis de usar. Programadores Python estão achando muitos usos para blocos `with` além do fechamento automático de arquivos.

A cláusula `else` não tem relação nenhuma com `with`. Mas esta é a Parte V, e não encontrei outro lugar para discuti-la; não quis escrever um capítulo de uma página, então aqui está ela.

Vamos analisar o tópico menor para chegar ao verdadeiro assunto deste capítulo.

¹ Apresentação na PyCon US 2013: “What Makes Python Awesome” (O que faz Python ser fantástico, <http://pyvideo.org/video/1669/keynote-3>); a parte sobre `with` começa em 23min00s e termina em 26min15s.

Faça isso, então aquilo: blocos `else` além de `if`

Não é nenhum segredo, mas a cláusula `else` é um recurso subutilizado da linguagem; ela pode ser usada não só em comandos `if`, mas também em comandos `for`, `while` e `try`.

As semânticas de `for/else`, `while/else` e `try/else` estão intimamente relacionadas, mas são muito diferentes da semântica de `if/else`. Inicialmente a palavra `else`, na verdade, atrapalhava meu entendimento desses recursos, mas, em algum momento, acostumei-me a ela.

Veja as regras:

for

O bloco `else` executará somente quando o laço `for` executar até o final (ou seja, não executará se `for` for interrompido com um `break`).

while

O bloco `else` executará somente quando o laço `while` sair porque a condição se tornou falsa (ou seja, não executará quando `while` for interrompido com um `break`).

try

O bloco `else` executará somente quando nenhuma exceção for levantada no bloco `try`. A documentação oficial (<http://bit.ly/1MMa1YB>) também afirma que “exceções na cláusula `else` não são tratadas pelas cláusulas `except` anteriores”.

Em todos os casos, a cláusula `else` também será ignorada se uma exceção ou um comando `return`, `break` ou `continue` fizerem o controle sair do bloco principal do comando associado.



Acho `else` uma péssima escolha de palavra reservada em todos os casos, exceto com `if`. Ela implica uma alternativa excludente, como “execute este laço; caso contrário, faça aquilo”, mas a semântica de `else` em laços é o oposto disso: “execute este laço, então faça aquilo”. Isso sugere que `then` seria uma palavra reservada melhor – e também faria sentido no contexto de `try`: “Tente isso, então faça aquilo”. Contudo acrescentar novas palavras reservadas pode quebrar muito código existente, e Guido evita isso como se fosse uma praga.

Usar `else` com esses comandos muitas vezes deixa o código mais legível e evita o trabalho de definir flags de controle ou adicionar comandos `if` extras.

O uso de `else` em laços geralmente segue o padrão do trecho de código a seguir:

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```

No caso de blocos `try/except`, `else` pode parecer redundante à primeira vista. Afinal de contas, `after_call()` no trecho de código a seguir executará somente se `dangerous_call()` não levantar uma exceção, certo?

```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

No entanto fazer isso coloca `after_call()` dentro do bloco `try` sem um bom motivo. Por questões de clareza e corretude, o corpo de um bloco `try` deve ter apenas comandos que possam gerar as exceções esperadas. Assim é bem melhor:

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

Agora está claro que o bloco `try` protege contra possíveis erros em `dangerous_call()`, e não em `after_call()`. Também é mais óbvio que `after_call()` executará somente se nenhuma exceção for levantada no bloco `try`.

Em Python, `try/except` é comumente usado para controle de fluxo, e não só para tratamento de erros. Há até mesmo uma sigla/slogan para isso, documentada no glossário oficial de Python (<https://docs.python.org/3/glossary.html#term-eafp>):

EAFP

Easier to ask for forgiveness than permission (Mais fácil pedir perdão que permissão, ou MFPPP). Esse estilo comum de codar em Python pressupõe a existência de chaves ou atributos válidos e captura exceções se a suposição se provar falsa. Esse estilo limpo e rápido é caracterizado pela presença de muitos comandos `try` e `except`. A técnica opõe-se ao estilo *LBYL* comum a muitas outras linguagens como C.

O glossário então define *LBYL*:

LBYL

Look before you leap (Olhe antes de saltar, ou OAS). Esse estilo de codar testa explicitamente as pré-condições antes de fazer chamadas ou consultas. Esse estilo contrasta com a abordagem *EAFP* e é caracterizado pela presença de muitos comandos `if`. Em um ambiente com várias threads (multi-threaded), a abordagem *LBYL* tem o risco de introduzir uma condição de concorrência entre “the looking”

(o olhar) e “the leaping” (o saltar). Por exemplo, o código `if key in mapping: return mapping[key]` pode falhar se outra thread remover `key` de `mapping` após o teste, mas antes da consulta. Esse problema pode ser resolvido com travas (locks) ou pelo uso da abordagem EAFP.

Considerando o estilo EAFP, faz mais sentido ainda conhecer e usar bem os blocos `else` em comandos `try/except`.

Vamos agora discutir o tópico principal deste capítulo: o poderoso comando `with`.

Gerenciadores de contexto e blocos `with`

Objetos gerenciadores de contexto existem para controlar um comando `with`, assim como iteradores existem para controlar um comando `for`.

O comando `with` foi concebido para simplificar o padrão `try/finally`, que garante que alguma operação seja realizada após um bloco de código, mesmo que o bloco tenha sido interrompido por causa de uma exceção, um `return` ou uma chamada a `sys.exit()`. O código na cláusula `finally` normalmente libera um recurso crítico ou restaura algum estado anterior temporariamente alterado.

O protocolo de gerenciador de contexto é composto dos métodos `__enter__` e `__exit__`. No início de `with`, `__enter__` é chamado no objeto gerenciador de contexto. O papel da cláusula `finally` é desempenhado por uma chamada a `__exit__` no objeto gerenciador de contexto no final do bloco `with`.

O exemplo mais comum é garantir que um objeto-arquivo seja fechado. Veja o exemplo 15.1, que tem uma demonstração detalhada do uso de `with` para fechar um arquivo.

Exemplo 15.1 – Demonstração de um objeto-arquivo como gerenciador de contexto

```
>>> with open('mirror.py') as fp: # ❶
...     src = fp.read(60) # ❷
...
>>> len(src)
60
>>> fp # ❸
<_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
>>> fp.closed, fp.encoding # ❹
(True, 'UTF-8')
>>> fp.read(60) # ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```


- ❶ `fp` é associado ao arquivo aberto porque o método `__enter__` do arquivo devolve `self`.
- ❷ Lê alguns dados de `fp`.
- ❸ A variável `fp` continua disponível.²
- ❹ Você pode ler os atributos do objeto `fp`.
- ❺ Mas não pode realizar E/S com `fp` porque, no final do bloco `with`, o método `TextIOWrapper.__exit__` é chamado e fecha o arquivo.

O primeiro comentário numerado no exemplo 15.1 destaca uma questão sutil, porém crucial: o objeto gerenciador de contexto é o resultado da avaliação da expressão após `with`, mas o valor associado à variável alvo (na cláusula `as`) é o resultado da chamada a `__enter__` no objeto gerenciador de contexto.

Por acaso, no exemplo 15.1, a função `open()` devolve uma instância de `TextIOWrapper` e seu método `__enter__` devolve `self`. Porém o método `__enter__` também pode devolver outro objeto, em vez de devolver o gerenciador de contexto.

Quando o controle de fluxo sai do bloco `with` de qualquer maneira, o método `__exit__` é chamado no objeto gerenciador de contexto, e não no objeto devolvido por `__enter__`.

A cláusula `as` do comando `with` é opcional. No caso de `open`, você sempre precisará dela para obter uma referência ao arquivo, mas alguns gerenciadores de contexto devolvem `None`, pois não têm nenhum objeto útil para devolver ao usuário.

O exemplo 15.2 mostra a operação de um gerenciador de contexto muito fútil, criado para enfatizar a distinção entre o gerenciador de contexto e o objeto devolvido pelo seu método `__enter__`.

Exemplo 15.2 – Testando a classe de gerenciador de contexto `LookingGlass`

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what: ❶
...     print('Alice, Kitty and Snowdrop') ❷
...     print(what)
...
pordwonS dna yttiK ,ecilA ❸
YKCOWREBBAJ
>>> what ❹
'JABBERWOCKY'
>>> print('Back to normal.') ❺
Back to normal.
```

² Blocos `with` não definem um novo escopo, como o fazem as funções e os módulos.

- ❶ O gerenciador de contexto é uma instância de `LookingGlass`; Python chama `__enter__` no gerenciador de contexto e o resultado é associado a `what`.
- ❷ Exibe uma `str` e, em seguida, o valor da variável alvo `what`.
- ❸ A saída de cada `print` é invertida.
- ❹ Agora o bloco `with` terminou. Podemos ver que o valor devolvido por `__enter__`, armazenado em `what`, é a string `'JABBERWOCKY'`.
- ❺ A saída agora não está mais invertida.

O exemplo 15.3 mostra a implementação de `LookingGlass`.

Exemplo 15.3 – `mirror.py`: código da classe de gerenciador de contexto `LookingGlass`

```
class LookingGlass:
    def __enter__(self): ❶
        import sys
        self.original_write = sys.stdout.write ❷
        sys.stdout.write = self.reverse_write ❸
        return 'JABBERWOCKY' ❹

    def reverse_write(self, text): ❺
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback): ❻
        import sys ❼
        sys.stdout.write = self.original_write ❽
        if exc_type is ZeroDivisionError: ❾
            print('Please DO NOT divide by zero!')
            return True ❿
        11
```

- ❶ Python chama `__enter__` sem outros argumentos além de `self`.
- ❷ Guarda o método `sys.stdout.write` original em um atributo de instância para usar depois.
- ❸ Faz um monkey-patch em `sys.stdout.write`, substituindo-o com nosso próprio método.
- ❹ Devolve a string `'JABBERWOCKY'` só para que tenhamos algo para colocar na variável alvo `what`.
- ❺ Nosso substituto para `sys.stdout.write` inverte o argumento `text` e chama a implementação original.

- ⑥ Python chama `__exit__` com `None`, `None`, `None` se tudo correr bem; se uma exceção for levantada, os três argumentos receberão os dados da exceção, conforme descrito mais adiante.
- ⑦ É barato importar um módulo novamente, pois Python o coloca em cache.
- ⑧ Restaura o método original em `sys.stdout.write`.
- ⑨ Se a exceção for diferente de `None` e seu tipo for `ZeroDivisionError`, exibe uma mensagem...
- ⑩ ... e devolve `True` para informar o interpretador que a exceção foi tratada.
- ⑪ Se `__exit__` devolver `None` ou algo diferente de `True`, qualquer exceção levantada no bloco `with` será propagada.



Quando aplicações de verdade assumem o controle da saída-padrão, com frequência, elas substituem `sys.stdout` por outro objeto arquivo ou similar por um tempo e, em seguida, restauram o original. O gerenciador de contexto `contextlib.redirect_stdout` (<http://bit.ly/1MM7Sw6>) faz exatamente isso; basta passar-lhe o objeto-arquivo ou similar que ficará no lugar de `sys.stdout`.

O interpretador chama o método `__enter__` sem argumentos – além do `self` implícito. Os três argumentos passados para `__exit__` são:

`exc_type`

A classe da exceção (por exemplo, `ZeroDivisionError`).

`exc_value`

A instância da exceção. Às vezes, parâmetros passados para o construtor da exceção – por exemplo, a mensagem de erro – podem ser encontrados em `exc_value.args`.

`traceback`

Um objeto `traceback`.³

Para ver como um gerenciador de contexto funciona em detalhes, observe o exemplo 15.4, no qual usamos `LookingGlass` fora de um bloco `with`, de modo que podemos chamar manualmente seus métodos `__enter__` e `__exit__`.

³ Os três argumentos recebidos por `self` são exatamente o que você terá se chamar `sys.exc_info()` (<http://bit.ly/1MM82Uc>) no bloco `finally` de um comando `try/finally`. Isso faz sentido, considerando que o comando `with` tem o propósito de substituir a maioria dos usos de `try/finally`, e frequentemente era preciso chamar `sys.exc_info()` para determinar a ação de limpeza necessária.

Exemplo 15.4 – Exercitando LookingGlass sem um bloco with

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass() ❶
>>> manager
<mirror.LookingGlass object at 0x2a578ac>
>>> monster = manager.__enter__() ❷
>>> monster == 'JABBERWOCKY' ❸
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager
>ca875a2x0 ta tcejbo ssalGgnikool.rorrim<
>>> manager.__exit__(None, None, None) ❹
>>> monster
'JABBERWOCKY'
```

- ❶ Instancia e inspeciona a instância manager.
- ❷ Chama o método `__enter__()` do gerenciador de contexto e armazena o resultado em `monster`.
- ❸ `monster` é a string `'JABBERWOCKY'`. O identificador `True` aparece invertido porque toda saída por meio de `stdout` passa pelo método `write` que alteramos em `__enter__`.
- ❹ Chama `manager.__exit__` para restaurar o `stdout.write` anterior.

Gerenciadores de contexto são um recurso razoavelmente novo e, aos poucos, mas seguramente, a comunidade Python vem encontrando usos novos e criativos para eles. Alguns exemplos da biblioteca-padrão são:

- administrar transações no módulo `sqlite3`; veja a seção “12.6.73. Using the connection as a context manager” (Usando a conexão como um gerenciador de contexto, <http://bit.ly/1MM89PC>).
- segurar travas, condições e semáforos no código de `threading`; veja a seção “17.1.10. Using locks, conditions, and semaphores in the with statement” (Usando travas, condições e semáforos no comando `with`, <http://bit.ly/1MM8guy>).
- configurar ambientes para operações aritméticas com objetos `Decimal`; veja a documentação de `decimal.localcontext` (<http://bit.ly/1MM8eTw>).
- aplicar patches temporários em objetos para testes; veja a função `unittest.mock.patch` (<http://bit.ly/1MM8imk>).

A biblioteca-padrão também inclui os utilitários de `contextlib`, discutidos a seguir.

Utilitários de contextlib

Antes de desenvolver suas próprias classes de gerenciador de contexto, dê uma olhada na seção “29.6 contextlib — Utilities for with-statement contexts” (contextlib – utilitários para contextos de comandos with, <http://bit.ly/1HGqZpf>) em *The Python Standard Library* (Biblioteca-Padrão de Python). Além do já mencionado `redirect_stdout`, o módulo `contextlib` inclui classes e outras funções com aplicações mais amplas:

`closing`

Uma função para criar gerenciadores de contexto a partir de objetos que ofereçam um método `close()`, mas não implementem o protocolo `__enter__/_exit__`.

`suppress`

Um gerenciador de contexto para ignorar temporariamente exceções especificadas.

`@contextmanager`

Um decorador que permite criar um gerenciador de contexto a partir de uma função geradora simples, em vez de criar uma classe e implementar o protocolo.

`ContextDecorator`

Uma classe-base para definir gerenciadores de contexto baseados em classe que também podem ser usados como decoradores de função, executando a função toda em um contexto gerenciado.

`ExitStack`

Um gerenciador de contexto que permite entrar em um número variável de gerenciadores de contexto. Quando o bloco `with` termina, `ExitStack` chama os métodos `__exit__` dos gerenciadores de contexto empilhados em ordem LIFO (último que entrou é o primeiro que sai). Use essa classe quando não souber de antemão em quantos gerenciadores de contexto você deve entrar em seu bloco `with`; por exemplo, quando abrir todos os arquivos de uma lista qualquer de arquivos ao mesmo tempo.

O utilitário mais amplamente usado entre esses certamente é o decorador `@contextmanager`, portanto merece mais atenção. Esse decorador também é intrigante, pois mostra um uso não relacionado à iteração para o comando `yield`. Isso prepara o caminho para o conceito de corrotina, que será o tema do próximo capítulo.

Usando @contextmanager

O decorador `@contextmanager` reduz o código repetitivo para criar um gerenciador de contexto: em vez de escrever uma classe completa com métodos `__enter__`/`__exit__`, basta implementar um gerador com um único `yield` que produza o que você quer que o método `__enter__` devolva.

Em um gerador decorado com `@contextmanager`, `yield` é usado para separar o corpo da função em duas partes: tudo que estiver antes de `yield` será executado no início do bloco `while`, quando o interpretador chamar `__enter__`; o código após `yield` será executado quando `__exit__` for chamado no final do bloco.

Veja um exemplo. O exemplo 15.5 substitui a classe `LookingGlass` do exemplo 15.3 por uma função geradora.

Exemplo 15.5 – `mirror_gen.py`: um gerenciador de contexto implementado com um gerador

```
import contextlib
```

```
@contextlib.contextmanager ❶
def looking_glass():
    import sys
    original_write = sys.stdout.write ❷

    def reverse_write(text): ❸
        original_write(text[::-1])

    sys.stdout.write = reverse_write ❹
    yield 'JABBERWOCKY' ❺
    sys.stdout.write = original_write ❻
```

- ❶ Aplica o decorador `contextmanager`.
- ❷ Preserva o método `sys.stdout.write` original.
- ❸ Define a função `reverse_write` personalizada; `original_write` estará disponível na closure.
- ❹ Substitui `sys.stdout.write` por `reverse_write`.
- ❺ Produz o valor que será associado à variável alvo na cláusula `as` do comando `with`. A função faz uma pausa nesse ponto, enquanto o corpo de `with` executa.
- ❻ Quando o controle deixa o bloco `with` de qualquer maneira, a execução continua após `yield`; nesse caso, o `sys.stdout.write` original é restaurado.

O exemplo 15.6 mostra a função `looking_glass` em ação.

Exemplo 15.6 – Testando a função de gerenciamento de contexto `looking_glass`

```
>>> from mirror_gen import looking_glass
>>> with looking_glass() as what: ❶
...     print('Alice, Kitty and Snowdrop')
...     print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
```

❶ A única diferença em relação ao exemplo 15.2 é o nome do gerenciador de contexto: `looking_glass` em vez de `LookingGlass`.

Essencialmente, o decorador `contextlib.contextmanager` encapsula a função em uma classe que implementa os métodos `__enter__` e `__exit__`.⁴

O método `__enter__` dessa classe faz o seguinte:

1. chama a função geradora e armazena o objeto gerador – vamos chamá-lo de `gen`.
2. chama `next(gen)` para executá-lo até a palavra reservada `yield`.
3. devolve o valor produzido por `next(gen)` para que possa ser associado a uma variável-alvo em `with/as`.

Quando o bloco `with` terminar, o método `__exit__`:

1. verificará se uma exceção foi passada como `exc_type`; se tiver sido, `gen.throw(exception)` será chamado, fazendo a exceção ser levantada na linha com `yield`, dentro do corpo da função geradora.
2. caso contrário, `next(gen)` será chamado, retomando a execução do corpo da função geradora após `yield`.

O exemplo 15.5 tem uma séria deficiência: se uma exceção for levantada no corpo do bloco `with`, o interpretador Python a capturará e a levantará novamente na expressão `yield` em `looking_glass`. Mas não há nenhum tratamento de erro ali, portanto a função `looking_glass` será interrompida sem restaurar o método `sys.stdout.write` original, deixando o sistema em um estado inválido.

O exemplo 15.7 acrescenta um tratamento especial para a exceção `ZeroDivisionError`, tornando-a funcionalmente equivalente ao exemplo 15.3 baseado em classe.

⁴ A classe propriamente dita chama-se `_GeneratorContextManager`. Se quiser ver como ela funciona exatamente, consulte seu código-fonte (<http://bit.ly/1MM8AJf>) em `Lib/contextlib.py` na distribuição de Python 3.4.

Exemplo 15.7 – `mirror_gen_exc.py`: gerenciador de contexto baseado em gerador implementando tratamento de exceção – mesmo comportamento externo do exemplo 15.3

```
import contextlib

@contextlib.contextmanager
def looking_glass():
    import sys
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = '' ❶
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError: ❷
        msg = 'Please DO NOT divide by zero!'
    finally:
        sys.stdout.write = original_write ❸
        if msg:
            print(msg) ❹
```

- ❶ Cria uma variável para uma possível mensagem de erro; essa é a primeira alteração em relação ao exemplo 15.5.
- ❷ Trata `ZeroDivisionError` definindo uma mensagem de erro.
- ❸ Desfaz o monkey-patching de `sys.stdout.write`.
- ❹ Exibe uma mensagem de erro se estiver definida.

Lembre-se de que o método `__exit__` diz ao interpretador que tratou a exceção devolvendo `True`; nesse caso, o interpretador suprime a exceção. Por outro lado, se `__exit__` não devolver explicitamente um valor, o interpretador obterá o `None` usual e propagará a exceção. Com `@contextmanager`, o comportamento-padrão é invertido: o método `__exit__` fornecido pelo decorador supõe que qualquer exceção enviada ao gerador é tratada e deve ser suprimida.⁵ Você deve explicitamente levantar uma exceção novamente na função decorada se não quiser que `@contextmanager` a suprima.⁶

5 A exceção é enviada para o gerador usando o método `throw`, discutido na seção “Término de corrotinas e tratamento de exceção” na página 524.

6 Essa convenção foi adotada porque, quando os gerenciadores de contexto foram criados, os geradores não podiam devolver valores (com `return`), somente produzi-los (com `yield`). Agora eles podem fazer isso, conforme explicado na seção “Devolvendo um valor a partir de uma corrotina” na página 528. Como você verá, devolver um valor a partir de um gerador envolve uma exceção.



Ter um `try/finally` (ou um bloco `with`) em torno de `yield` é um preço inevitável pelo uso de `@contextmanager`, pois você nunca sabe o que os usuários de seu gerenciador de contexto farão dentro do bloco `with` deles.^(*)

^(*) Essa dica foi literalmente copiada de um comentário de Leonardo Rochael, um dos revisores técnicos deste livro. Muito bem colocado, Leo!

Um exemplo interessante de `@contextmanager` do mundo real, fora da biblioteca-padrão, é o gerenciador de contexto para reescrita in-place de arquivos (<http://bit.ly/1MM96aR>) de Martijn Pieters. O exemplo 15.8 mostra como ele é usado.

Exemplo 15.8 – Um gerenciador de contexto para reescrever arquivos in-place

```
import csv

with inplace(csvfilename, 'r', newline='') as (infh, outfh):
    reader = csv.reader(infh)
    writer = csv.writer(outfh)

    for row in reader:
        row += ['new', 'columns']
        writer.writerow(row)
```

A função `inplace` é um gerenciador de contexto que oferece dois handles – `infh` e `outfh` no exemplo – para o mesmo arquivo, permitindo ao código ler e escrever no arquivo ao mesmo tempo. É mais fácil de usar que a função `fileinput.input` (<http://bit.ly/1HGr6Sq>) da biblioteca-padrão (que também oferece um gerenciador de contexto, a propósito).

Se quiser estudar o código-fonte de `inplace` de Martijn (listado no post em <http://bit.ly/1MM96aR>), procure a palavra reservada `yield`: tudo que estiver antes dela lida com configuração do contexto, que implica criar um arquivo de backup, em seguida abrir os arquivos e produzir referências para os handles do arquivo para leitura e escrita, que serão devolvidos pela chamada a `__enter__`. O processamento de `__exit__` após `yield` fecha os handles do arquivo e restaura o arquivo do backup se algo der errado.

Observe que o uso de `yield` em um gerador com o decorador `@contextmanager` não tem nada a ver com iteração. Nos exemplos mostrados nesta seção, a função geradora funciona mais como uma corrotina: um procedimento que executa até um ponto e, em seguida, é suspenso para permitir a execução do código do cliente até o cliente querer que a corrotina prossiga com sua tarefa. O capítulo 16 é todo dedicado às corrotinas.

Resumo do capítulo

Este capítulo começou de forma tranquila com a discussão sobre blocos `else` em comandos `for`, `while` e `try`. Depois que você se acostuma com o significado peculiar da cláusula `else` nesses comandos, acredito que ela pode deixar suas intenções mais claras.

Em seguida, discutimos os gerenciadores de contexto e o significado do comando `with`, avançando rapidamente para além de seu uso comum de fechar arquivos abertos automaticamente. Implementamos um gerenciador de contexto personalizado: a classe `LookingGlass` com os métodos `__enter__`/`__exit__`, e vimos como tratar exceções no método `__exit__`. Um ponto fundamental que Raymond Hettinger destacou em sua apresentação na PyCon US 2013 foi que `with` não serve apenas para gerenciamento de recursos, mas é uma ferramenta para fatorar códigos comuns para fazer e desfazer configurações, ou qualquer par de operações que precisem ser feitas antes e depois de outro procedimento [slide 21, What Makes Python Awesome? (O que faz Python ser incrível?, <http://bit.ly/1MM9pCm>)].

Por fim, analisamos funções do módulo `contextlib` da biblioteca-padrão. Uma delas, o decorador `@contextmanager`, possibilitou implementar um gerenciador de contexto usando um gerador simples com um `yield` – uma solução mais enxuta que codar uma classe com pelo menos dois métodos. Reimplementamos `LookingGlass` como uma função geradora `looking_glass` e discutimos o tratamento de exceções quando usamos `@contextmanager`.

O decorador `@contextmanager` é uma ferramenta elegante e prática, que reúne três recursos característicos de Python: um decorador de função, um gerador e o comando `with`.

Leituras complementares

O capítulo 8 – “Compound Statements” (Comandos compostos, <http://bit.ly/1MMa1YB>) – de *The Python Language Reference* (Guia de referência à linguagem Python) diz praticamente tudo que há para dizer sobre cláusulas `else` em comandos `if`, `for`, `while` e `try`. Quanto ao uso pythônico de `try/except`, com ou sem `else`, Raymond Hettinger tem uma resposta brilhante à pergunta “Is it a good practice to use try-except-else in Python?” (Usar try-except-else em Python é uma boa prática?, <http://bit.ly/1MMa2Mp>) em StackOverflow. O livro *Python in a Nutshell*, 2E (O’Reilly, <http://shop.oreilly.com/product/9780596100469.do>) de Alex Martelli tem um capítulo sobre exceções com uma excelente discussão sobre o estilo EAFP, dando o crédito à pioneira em computação Grace Hopper por ter cunhado a frase “It’s easier to ask forgiveness than permission” (Mais fácil pedir perdão que permissão, ou MFPPP).

O capítulo 4 – “Built-in Types” (Tipos embutidos) – de *Python Standard Library* (Biblioteca-Padrão de Python) tem uma seção dedicada a *Context Manager Types* (Tipos para gerenciadores de contexto, <http://bit.ly/1MMacTS>). Os métodos especiais `__enter__`/`__exit__` também estão documentados em *The Python Language Reference* na seção “3.3.8. With Statement Context Managers” (Gerenciadores de contexto do comando `with`, <http://bit.ly/1MMab2e>). Gerenciadores de contexto foram introduzidos na PEP 343 – *The “with” Statement* (O comando “with”, <https://www.python.org/dev/peps/pep-0343/>). Essa PEP não tem uma leitura fácil, pois gasta muito tempo discutindo casos extremos (corner cases) e argumentando contra propostas alternativas. É a natureza das PEPs.

Raymond Hettinger destacou o comando `with` como um “recurso vencedor da linguagem” em sua apresentação na PyCon US 2013 (<http://bit.ly/1MM9pCm>). Ele também mostrou algumas aplicações interessantes de gerenciadores de contexto em sua palestra “Transforming Code into Beautiful, Idiomatic Python” (Transformando códigos em Python bonito e idiomático, <http://bit.ly/1MMagmB>) na mesma conferência.

O blog post “The Python with Statement by Example” (O comando `with` de Python por meio de exemplos, <http://bit.ly/1MMakmm>) de Jeff Preshing é interessante pelos exemplos que usam gerenciadores de contexto com a biblioteca gráfica `pycairo`.

Beazley e Jones inventaram gerenciadores de contexto para propósitos bem diferentes em seu livro *Python Cookbook*, 3E (O’Reilly, <http://shop.oreilly.com/product/0636920027072.do>)⁷. A “Recipe 8.3. Making Objects Support the Context-Management Protocol” (Fazer os objetos suportarem o protocolo de gerenciamento de contexto) implementa uma classe `LazyConnection` cujas instâncias são gerenciadores de contexto que abrem e fecham conexões de rede automaticamente em blocos `with`. A “Recipe 9.22. Defining Context Managers the Easy Way” (Definir gerenciadores de contexto da maneira fácil) introduz um gerenciador de contexto para medir o tempo de execução de códigos e outro para fazer mudanças transacionais em um objeto `list`; no bloco `with`, uma cópia de trabalho da instância de `list` é feita e todas as alterações são aplicadas a essa cópia de trabalho. A cópia de trabalho substitui a lista original somente quando o bloco `with` termina sem exceção. Simples e engenhoso.

⁷ N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

Ponto de vista

Fatorando o pão

Em sua apresentação “What Makes Python Awesome” (O que faz Python ser fantástico, <http://pyvideo.org/video/1669/keynote-3>) na PyCon US 2013, Raymond Hettinger diz que, na primeira vez em que viu a proposta para o comando `with`, achou que era “um pouco arcano”. Inicialmente, eu tive uma reação parecida. As PEPs muitas vezes são difíceis de ler, e a PEP 343 é típica nesse aspecto.

Então – contou Hettinger – ele teve um insight: sub-rotinas são a invenção mais importante da história das linguagens de computador. Se você tiver sequências de operações como `A;B;C` e `P;B;Q`, poderá fatorar a operação `B` e extraí-la para uma sub-rotina. É como fatorar o recheio de um sanduíche: usar atum com pães diferentes. Mas e se você quisesse fatorar o pão para fazer sanduíches com pão integral usando um recheio diferente a cada vez? É isso que o comando `with` oferece. Ele é o complemento da sub-rotina. Hettinger prosseguiu dizendo:

O comando `with` é muito importante. Incentivo você a sair por aí, pegar essa ponta do iceberg e explorá-lo melhor. É provável que você possa fazer coisas muito profundas com o comando `with`. Os melhores usos dele ainda não foram descobertos. Se fizermos bom uso desse comando, acredito que ele será imitado em outras linguagens, e todas as futuras linguagens o incluirão. Você pode participar da descoberta de algo quase tão profundo quanto a invenção da própria sub-rotina.

Hettinger admite que está exagerando nas qualidades do comando `with`. Apesar disso, é um recurso muito útil. Quando ele usou a analogia do sanduíche para explicar como `with` é o complemento da sub-rotina, muitas possibilidades se abriram em minha mente.

Se precisar convencer alguém de que Python é sensacional, assista à apresentação de Hettinger. O trecho sobre gerenciadores de contexto está entre 23min00s e 26min15s. Mas a apresentação toda é excelente.

CAPÍTULO 16

Corrotinas

Se livros sobre Python são alguma indicação, este [as corrotinas] seria o recurso mais mal documentado, obscuro e aparentemente inútil de Python.

— David Beazley

Autor de livros sobre Python

Encontramos dois significados principais para o verbo “to yield” em dicionários: *produce* (produzir) ou *give way* (ceder a vez). Ambos os sentidos se aplicam em Python quando usamos a palavra reservada `yield` em um gerador. Uma linha como `yield item` produz um valor que é recebido por quem chama `next(...)` e cede o controle, suspendendo a execução do gerador para que o chamador (caller) possa prosseguir, até que esteja pronto para consumir outro valor chamando `next()` novamente. O chamador extrai valores do gerador.

Uma corrotina, do ponto de vista sintático, é como um gerador: apenas uma função com a palavra reservada `yield` em seu corpo. Entretanto, em uma corrotina, `yield` normalmente aparece do lado direito de uma expressão (por exemplo, `datum = yield`), e pode ou não produzir um valor – se não houver nenhuma expressão após a palavra reservada `yield`, o gerador produz `None`. A corrotina pode receber dados do chamador, que usa `co.send(datum)` em vez de `next(co)` para alimentar a corrotina. Normalmente, o chamador envia valores à corrotina.

É até possível que não haja troca de dados com a palavra reservada `yield`. Independentemente do fluxo de dados, `yield` é um dispositivo de controle de fluxo que pode ser usado para implementar multitarefa cooperativa (cooperative multitasking): cada corrotina cede o controle a um escalonador central para que outras corrotinas possam ser ativadas.

Quando começar a pensar em `yield` principalmente em termos de controle de fluxo, você estará com a mentalidade adequada para entender as corrotinas.

As corrotinas em Python são o produto de uma série de melhorias nas humildes funções geradoras que vimos até agora no livro. Seguir a evolução das corrotinas em Python ajuda a entender seus recursos em etapas, cada uma com mais funcionalidades e complexidade.

Após uma rápida visão geral de como os geradores passaram a atuar como corrotinas, passaremos para a parte mais importante do capítulo. Então veremos:

- o comportamento e os estados de um gerador funcionando como uma corrotina;
- preparação (priming) automática de uma corrotina com um decorador;
- como o chamador pode controlar uma corrotina por meio dos métodos `.close()` e `.throw(_)` do objeto gerador;
- como as corrotinas podem devolver valores no término;
- uso e semântica da nova sintaxe `yield from`;
- um caso de uso; corrotinas para gerenciar atividades concorrentes em uma simulação.

Como as corrotinas evoluíram a partir de geradores

A infraestrutura para corrotinas surgiu na *PEP 342 – Coroutines via Enhanced Generators* (Corrotinas por meio de geradores melhorados, <https://www.python.org/dev/peps/pep-0342/>), implementada em Python 2.5 (2006): desde então, a palavra reservada `yield` pode ser usada em uma expressão, e o método `.send(value)` foi adicionado à API de geradores. Usando `.send(_)`, quem chama o gerador pode enviar dados que, então, tornam-se o valor da expressão `yield` na função geradora. Isso permite que um gerador seja usado como uma corrotina: um procedimento que possibilita a colaboração com o chamador, entregando-lhe e recebendo valores dele.

Além de `.send(_)`, a PEP 342 também adicionou os métodos `.throw(...)` e `.close()` que, respectivamente, permitem ao chamador lançar uma exceção a ser tratada no gerador e terminá-lo. Esses recursos serão discutidos na próxima seção e na seção “Término de corrotinas e tratamento de exceção” na página 524.

Um passo evolucionário importante veio com a *PEP 380 – Syntax for Delegating to a Subgenerator* (Sintaxe para delegar a um subgerador, <https://www.python.org/dev/peps/pep-0380/>), implementada em Python 3.3 (2012). A PEP 380 fez duas alterações de sintaxe em funções geradoras para torná-las mais úteis como corrotinas:

- Um gerador agora pode devolver um valor (com `return`); antes, fornecer um valor ao comando `return` em um gerador levantava `SyntaxError`.

- A sintaxe `yield from` permite que geradores complexos sejam refatorados em geradores menores e aninhados, ao mesmo tempo que evita muito código repetitivo anteriormente necessário para um gerador delegar a subgeradores.

Essas últimas alterações serão discutidas nas seções “Devolvendo um valor a partir de uma corrotina” na página 528 e “Usando `yield from`” na página 530.

Vamos seguir a tradição estabelecida em *Python fluente* e começar com alguns fatos e exemplos básicos e, em seguida, passaremos para funcionalidades cada vez mais complexas.

Comportamento básico de um gerador usado como corrotina

O exemplo 16.1 mostra o comportamento de uma corrotina.

Exemplo 16.1 – Demonstração mais simples possível de uma corrotina em ação

```
>>> def simple_coroutine(): # ❶
...     print('-> coroutine started')
...     x = yield # ❷
...     print('-> coroutine received:', x)
...
>>> my_coro = simple_coroutine()
>>> my_coro # ❸
<generator object simple_coroutine at 0x100c2be10>
>>> next(my_coro) # ❹
-> coroutine started
>>> my_coro.send(42) # ❺
-> coroutine received: 42
Traceback (most recent call last): # ❻
...
StopIteration
```

- ❶ Uma corrotina é definida como uma função geradora: com `yield` em seu corpo.
- ❷ `yield` é usado em uma expressão; quando a corrotina é projetada somente para receber dados do cliente, ela produz `None` – isso está implícito porque não há nenhuma expressão à direita da palavra reservada `yield`.
- ❸ Como já ocorre com geradores, você chama a função para obter um objeto gerador.
- ❹ A primeira chamada é `next(...)` porque o gerador ainda não começou, portanto não está esperando em um `yield` e não podemos enviar-lhe nenhum dado inicialmente.
- ❺ Essa chamada faz `yield` no corpo da corrotina ser avaliado com 42; agora a corrotina retoma a execução e avança até o próximo `yield` ou até terminar.

- ❻ Nesse caso, o controle flui para o final do corpo da corrotina, o que faz o mecanismo do gerador levantar `StopIteration`, como é feito normalmente.

Uma corrotina pode estar em um de quatro estados. Você pode determinar o estado atual usando a função `inspect.getgeneratorstate(...)`, que devolve uma das strings a seguir:

`'GEN_CREATED'`

Esperando para iniciar a execução.

`'GEN_RUNNING'`

No momento, está sendo executada pelo interpretador.¹

`'GEN_SUSPENDED'`

No momento, está suspensa em uma expressão `yield`.

`'GEN_CLOSED'`

A execução terminou.

Como o argumento do método `send` será o valor da expressão `yield` pendente, isso implica que você só poderá fazer uma chamada como `my_coro.send(42)` se a corrotina estiver suspensa no momento. Mas esse não será o caso se a corrotina ainda não tiver sido ativada – se o seu estado for `'GEN_CREATED'`. É por isso que a primeira ativação de uma corrotina é sempre feita com `next(my_coro)` – você também pode chamar `my_coro.send(None)`, que o efeito será o mesmo.

Se criar um objeto corrotina e tentar enviar-lhe um valor diferente de `None` imediatamente, veja o que acontece:

```
>>> my_coro = simple_coroutine()
>>> my_coro.send(1729)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

Observe a mensagem de erro, ela é bem clara: “não se pode enviar um valor diferente de `None` para um gerador que acabou de iniciar”.

A chamada inicial `next(my_coro)` muitas vezes é descrita como “preparação” (priming) da corrotina (ou seja, fazê-la avançar até o primeiro `yield` deixando-a pronta para ser usada como uma corrotina ativa).

Para ter uma noção melhor do comportamento de uma corrotina, um exemplo que produz valor mais de uma vez será útil. Veja o exemplo 16.2.

¹ Você só verá esse estado em uma aplicação com várias threads (multithreaded) – ou se o objeto gerador chamar `getgeneratorstate` em si mesmo, o que não é muito útil.

Exemplo 16.2 – Uma corrotina que produz valor duas vezes

```

>>> def simple_coro2(a):
...     print('-> Started: a =', a)
...     b = yield a
...     print('-> Received: b =', b)
...     c = yield a + b
...     print('-> Received: c =', c)
...
>>> my_coro2 = simple_coro2(14)
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(my_coro2) ❶
'GEN_CREATED'
>>> next(my_coro2) ❷
-> Started: a = 14
14
>>> getgeneratorstate(my_coro2) ❸
'GEN_SUSPENDED'
>>> my_coro2.send(28) ❹
-> Received: b = 28
42
>>> my_coro2.send(99) ❺
-> Received: c = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> getgeneratorstate(my_coro2) ❻
'GEN_CLOSED'

```

- ❶ `inspect.getgeneratorstate` informa `GEN_CREATED` (ou seja, a corrotina não iniciou).
- ❷ Avança a corrotina até o primeiro `yield`, exibindo a mensagem `-> Started: a = 14` e, em seguida, produz o valor de `a` e fica suspensa à espera do valor que será atribuído a `b`.
- ❸ `getgeneratorstate` informa `GEN_SUSPENDED` (ou seja, a corrotina está parada em uma expressão `yield`).
- ❹ Envia o número 28 à corrotina suspensa; a expressão `yield` é avaliada como 28 e esse número é associado a `b`. A mensagem `-> Received: b = 28` é exibida, o valor de `a + b` é produzido (42) e a corrotina é suspensa à espera do valor que será atribuído a `c`.
- ❺ Envia o número 99 à corrotina suspensa; a expressão `yield` é avaliada como 99 e o número é associado a `c`. A mensagem `-> Received: c = 99` é exibida; em seguida, a corrotina termina, fazendo o objeto gerador levantar `StopIteration`.
- ❻ `getgeneratorstate` informa `GEN_CLOSED` (ou seja, a execução da corrotina terminou).

É muito importante entender que a execução da corrotina é suspensa exatamente na palavra reservada `yield`. Como mencionamos antes, em um comando de atribuição, o código à direita de `=` é avaliado antes de a atribuição propriamente dita ocorrer. Isso quer dizer que em uma linha como `b = yield a`, o valor de `b` só será definido quando a corrotina for posteriormente ativada pelo código do cliente. É preciso um pouco de esforço para acostumar-se com esse fato, mas entendê-lo é essencial para compreender o uso de `yield` em programação assíncrona, como veremos depois.

A execução da corrotina `simple_coro2` pode ser dividida em três fases, como mostra a figura 16.1:

1. `next(my_coro2)` exibe a primeira mensagem e executa até `yield a`, produzindo o número 14.
2. `my_coro2.send(28)` atribui 28 a `b`, exibe a segunda mensagem e executa até `yield a + b`, produzindo o número 42.
3. `my_coro2.send(99)` atribui 99 a `c`, exibe a terceira mensagem e a corrotina termina.

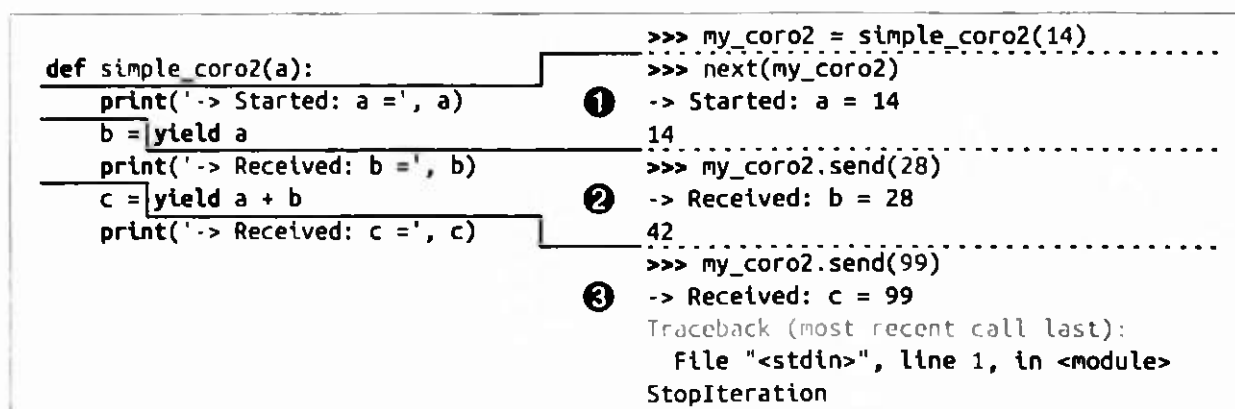


Figura 16.1 – Três fases na execução da corrotina `simple_coro2` (observe que cada fase termina em uma expressão `yield`, e a próxima fase começa exatamente na mesma linha, quando o valor da expressão `yield` é atribuído a uma variável).

Vamos agora considerar um exemplo um pouco mais sofisticado de corrotina.

Exemplo: corrotina para calcular uma média cumulativa

Quando discutimos closures no capítulo 7, estudamos objetos para calcular uma média cumulativa: o exemplo 7.8 mostrou uma classe simples e o exemplo 7.14 apresentou uma função de ordem superior produzindo uma closure para preservar as variáveis `total` e `count` entre as chamadas. O exemplo 16.3 mostra como fazer o mesmo com uma corrotina.²

² Esse exemplo foi inspirado em um trecho de código de Jacob Holm na lista Python-ideas de uma mensagem cujo título é “Yield-From: Finalization guarantees” (Yield-from: garantias de finalização, <http://bit.ly/1MMc9zy>). Algumas variações apareceram depois na discussão, e Holm explicou melhor seu raciocínio na mensagem 003912 (<http://bit.ly/1MMcano>).

Exemplo 16.3 – `coroaverager0.py`: código de uma corrotina para cálculo de média cumulativa

```
def averager():  
    total = 0.0  
    count = 0  
    average = None  
    while True: ❶  
        term = yield average ❷  
        total += term  
        count += 1  
        average = total/count
```

- ❶ Esse laço infinito significa que essa corrotina continuará aceitando valores e produzindo resultados enquanto o chamador enviar valores. Essa corrotina só terminará quando o código do cliente chamar `.close()` nela ou quando for eliminada pelo coletor de lixo por não haver mais referências a ela.
- ❷ O comando `yield` aqui é usado para suspender a corrotina, produzir um resultado a quem chamou e – mais tarde – obter um valor enviado pelo chamador à corrotina, que retoma seu laço infinito.

A vantagem de usar uma corrotina é que `total` e `count` podem ser variáveis locais simples: nenhum atributo de instância ou closure é necessário para preservar o contexto entre as chamadas. O exemplo 16.4 contém doctests para mostrar a corrotina `averager` em ação.

Exemplo 16.4 – `coroaverager0.py`: doctest para a corrotina do exemplo 16.3 que calcula uma média cumulativa

```
>>> coro_avg = averager() ❶  
>>> next(coro_avg) ❷  
>>> coro_avg.send(10) ❸  
10.0  
>>> coro_avg.send(30)  
20.0  
>>> coro_avg.send(5)  
15.0
```

- ❶ Cria o objeto corrotina.
- ❷ Prepara-o chamando `next`.
- ❸ Agora estamos em operação: cada chamada a `.send(...)` produz a média atual.

No doctest (Exemplo 16.4), a chamada a `next(coro_avg)` faz a corrotina avançar até `yield`, produzindo o valor inicial de `average`, que é `None`, portanto não aparece no console. Nesse ponto, a corrotina fica suspensa no `yield`, esperando um valor ser enviado. A

linha `coro_avg.send(10)` fornece esse valor, fazendo a corrotina ser ativada, atribuindo o valor a `term`, atualizando as variáveis `total`, `count` e `average` e então iniciando outra iteração no laço `while`, que produz `average` e espera outro `term`.

O leitor atento pode estar ansioso para saber como a execução de uma instância de `averager` (por exemplo, `coro_avg`) pode ser terminada, pois seu corpo é um laço infinito. Discutiremos isso na seção “Término de corrotinas e tratamento de exceção” na página 524.

Porém, antes de discutir o término de corrotinas, vamos falar sobre como iniciá-las. Preparar uma corrotina antes de usá-la é uma tarefa necessária, porém fácil de esquecer. Para evitar isso, um decorador especial pode ser aplicado à corrotina. Um desses decoradores será apresentado a seguir.

Decoradores para preparação de corrotinas

Você não pode fazer muita coisa com uma corrotina sem prepará-la; é preciso sempre lembrar de chamar `next(my_coro)` antes de `my_coro.send(x)`. Para deixar mais conveniente o uso de uma corrotina, um decorador de preparação às vezes é usado. O decorador `coroutine` no exemplo 16.5 é um exemplo.³

Exemplo 16.5 – `coroutil.py`: decorador para preparação de corrotina

```
from functools import wraps

def coroutine(func):
    """Decorador: prepara `func` fazendo-a avançar até o primeiro `yield`"""
    @wraps(func)
    def primer(*args,**kwargs): ❶
        gen = func(*args,**kwargs) ❷
        next(gen) ❸
        return gen ❹
    return primer
```

- ❶ A função geradora decorada é substituída por essa função `primer` que, quando chamada, devolve o gerador preparado.
- ❷ Chama a função decorada para obter um objeto gerador.
- ❸ Prepara o gerador.
- ❹ Devolve-o.

³ Há muitos decoradores semelhantes publicados na Web. Esse foi adaptado da receita de *ActiveState Pipeline made of coroutines* (Pipeline feito de corrotinas, <http://bit.ly/1MMcuCx>) de Chaobin Tang, que, por sua vez, dá o crédito a David Beazley.

O exemplo 16.6 mostra o decorador `@coroutine` em uso. Compare com o exemplo 16.3.

Exemplo 16.6 – `coroaverager1.py`: doctest e código para uma corrotina de média cumulativa, que usa o decorador `@coroutine` do exemplo 16.5

```
"""
```

Uma corrotina para calcular uma média cumulativa

```
>>> coro_avg = averager() ❶
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(coro_avg) ❷
'GEN_SUSPENDED'
>>> coro_avg.send(10) ❸
10.0
>>> coro_avg.send(30)
20.0
>>> coro_avg.send(5)
15.0
```

```
"""
```

```
from coroutil import coroutine ❹
```

```
@coroutine ❺
```

```
def averager(): ❻
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield average
        total += term
        count += 1
        average = total/count
```

- ❶ Chama `averager()`, criando um objeto gerador que é preparado na função `primer` do decorador `coroutine`.
- ❷ `getgeneratorstate` informa `GEN_SUSPENDED`, indicando que a corrotina está pronta para receber um valor.
- ❸ Você pode começar a enviar valores imediatamente para `coro_avg`; é para isso que serve o decorador.
- ❹ Importa o decorador `coroutine`.
- ❺ Aplica-o à função `averager`.
- ❻ O corpo da função é exatamente o mesmo do exemplo 16.3.

Vários frameworks oferecem decoradores especiais projetados para trabalhar com corrotinas. Nem todos, na verdade, preparam a corrotina – alguns oferecem outros serviços, por exemplo, registrá-la em um laço de eventos. Um exemplo é o decorador `tornado.gen` da biblioteca de rede assíncrona Tornado (<http://bit.ly/1MMcGBF>).

A sintaxe `yield from` que veremos na seção “Usando `yield from`” na página 530 prepara automaticamente a corrotina chamada por ela, tornando-a incompatível com decoradores como `@coroutine` do exemplo 16.5. O decorador `asyncio.coroutine` da biblioteca-padrão de Python 3.4 foi projetado para funcionar com `yield from` e, sendo assim, não prepara a corrotina. Nós o discutiremos no capítulo 18.

Vamos agora nos concentrar em recursos essenciais de corrotinas: os métodos usados para terminá-las e lançar exceções nelas.

Término de corrotinas e tratamento de exceção

Uma exceção não tratada em uma corrotina se propaga a quem chamou o `next` ou o `send` que a disparou. O exemplo 16.7 tem um exemplo que usa a corrotina decorada `averager` do exemplo 16.6.

Exemplo 16.7 – Como uma exceção não tratada mata uma corrotina

```
>>> from coroaverager1 import averager
>>> coro_avg = averager()
>>> coro_avg.send(40) # ❶
40.0
>>> coro_avg.send(50)
45.0
>>> coro_avg.send('spam') # ❷
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +=: 'float' and 'str'
>>> coro_avg.send(60) # ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- ❶ Usando o `averager` decorado com `@coroutine`, podemos começar a enviar valores imediatamente.
- ❷ Enviar um valor não numérico causa uma exceção na corrotina.
- ❸ Como a exceção não foi tratada na corrotina, ela termina. Qualquer tentativa de reativá-la levantará `StopIteration`.

A causa do erro foi o envio de um valor 'spam' que não pôde ser somado à variável total da corrotina.

O exemplo 16.7 sugere uma maneira de terminar corrotinas: você pode usar `send` com algum valor de sentinela que diga à corrotina para sair. Constantes singleton embutidas como `None` e `Ellipsis` são valores convenientes para sentinelas. `Ellipsis` tem a vantagem de ser bastante incomum em streams de dados. Outro valor de sentinela que já vi sendo usado é `StopIteration` – a própria classe, e não uma instância dela (sem levantá-la). Em outras palavras, passando-a como `my_coro.send(StopIteration)`.

Desde Python 2.5, os objetos geradores têm dois métodos que permitem ao cliente disparar exceções dentro de uma corrotina – `throw` e `close`:

```
generator.throw(exc_type[, exc_value[, traceback]])
```

Faz a expressão `yield` em que o gerador estava parado levantar a exceção dada. Se a exceção for tratada pelo gerador, o fluxo avançará para o próximo `yield` e o valor produzido se tornará o valor da chamada a `generator.throw`. Se a exceção não for tratada pelo gerador, ela se propagará até o contexto de quem fez a chamada.

```
generator.close()
```

Faz a expressão `yield` em que o gerador estava parado levantar uma exceção `GeneratorExit`. Nenhum erro é informado a quem chamou se o gerador não tratar essa exceção nem levantar `StopIteration` – normalmente executando até terminar. Ao receber um `GeneratorExit`, o gerador não deve produzir um valor; caso contrário, um `RuntimeError` será levantado. Se qualquer outra exceção for levantada pelo gerador, ela se propagará até quem fez a chamada.



A documentação oficial dos métodos de objetos geradores está escondida em *The Python Language Reference* [Guia de referência à linguagem Python; veja a seção 6.2.9.1. *Generator-iterator methods* (Métodos de gerador-iterador, <https://docs.python.org/3/reference/expressions.html#generator-iterator-methods>)].

Vamos ver como `close` e `throw` controlam uma corrotina. O exemplo 16.8 lista a função `demo_exc_handling` usada nos exemplos seguintes.

Exemplo 16.8 – `coro_exc_demo.py`: código de teste para estudar o tratamento de exceção em uma corrotina

```
class DemoException(Exception):
    """Um tipo de exceção para a demonstração."""

def demo_exc_handling():
    print('-> coroutine started')
    while True:
        try:
```

```

        x = yield
    except DemoException: ❶
        print('*** DemoException handled. Continuing...')
    else: ❷
        print('-> coroutine received: {!r}'.format(x))
    raise RuntimeError('This line should never run.') ❸

```

❶ Tratamento especial para `DemoException`.

❷ Se não houver exceção, exibe o valor recebido.

❸ Essa linha jamais será executada.

A última linha do exemplo 16.8 jamais será executada, pois o laço infinito só pode ser interrompido por uma exceção não tratada, e isso faz a corrotina terminar imediatamente.

A operação normal de `demo_exc_handling` está no exemplo 16.9.

Exemplo 16.9 – Ativando e encerrando `demo_exc_handling` sem uma exceção

```

>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
>>> exc_coro.send(22)
-> coroutine received: 22
>>> exc_coro.close()
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(exc_coro)
'GEN_CLOSED'

```

Se `DemoException` for lançada na corrotina, ela será tratada e a corrotina `demo_exc_handling` continuará, como mostra o exemplo 16.10.

Exemplo 16.10 – Lançando `DemoException` em `demo_exc_handling` não a termina

```

>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
>>> exc_coro.throw(DemoException)
*** DemoException handled. Continuing...
>>> getgeneratorstate(exc_coro)
'GEN_SUSPENDED'

```

Por outro lado, se uma exceção não tratada for lançada na corrotina, ela parará – seu estado passará para 'GEN_CLOSED'. O exemplo 16.11 mostra isso.

Exemplo 16.11 – A corrotina termina se não puder tratar uma exceção lançada nela

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
>>> exc_coro.throw(ZeroDivisionError)
Traceback (most recent call last):
...
ZeroDivisionError
>>> getgeneratorstate(exc_coro)
'GEN_CLOSED'
```

Se for preciso executar algum código de limpeza independentemente de como a corrotina terminar, insira a parte relevante do corpo da corrotina em um bloco try/finally, como mostra o exemplo 16.12.

Exemplo 16.12 – coro_finally_demo.py: uso de try/finally para executar ações no término da corrotina

```
class DemoException(Exception):
    """Um tipo de exceção para a demonstração."""

def demo_finally():
    print('-> coroutine started')
    try:
        while True:
            try:
                x = yield
            except DemoException:
                print('*** DemoException handled. Continuing...')
            else:
                print('-> coroutine received: {!r}'.format(x))
    finally:
        print('-> coroutine ending')
```

Um dos principais motivos para a adição de yield from em Python 3.3 tem a ver com lançar exceções em corrotinas aninhadas. O outro motivo foi permitir que as corrotinas devolvessem valores de modo mais conveniente. Continue lendo para ver como isso é feito.

Devolvendo um valor a partir de uma corrotina

O exemplo 16.13 mostra uma variação da corrotina `averager` que devolve um resultado. Por motivos didáticos, ela não produz a média atual a cada ativação. Isso serve para enfatizar que algumas corrotinas não produzem nada interessante, mas são projetadas para devolver um valor no final, que, com frequência, é o resultado de algum valor acumulado.

O resultado devolvido por `averager` no exemplo 16.13 é uma `namedtuple` com o número de termos usados no cálculo da média (`count`) e `average`. Eu poderia ter devolvido apenas o valor de `average`, mas devolver uma tupla expõe outro dado interessante acumulado: o número de itens (`count`).

Exemplo 16.13 – `coroaverager2.py`: código de uma corrotina `averager` que devolve um resultado

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')

def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None:
            break ❶
        total += term
        count += 1
        average = total/count
    return Result(count, average) ❷
```

- ❶ Para devolver um valor, uma corrotina deve terminar normalmente; é por isso que essa versão de `averager` tem uma condição para sair de seu laço de acumulação.
- ❷ Devolve uma `namedtuple` com `count` e `average`. Antes de Python 3.3, devolver um valor em uma função geradora era considerado erro de sintaxe.

Para ver como esse novo `averager` funciona, podemos acioná-lo a partir do console, como mostra o exemplo 16.14.

Exemplo 16.14 – `coroaverager2.py`: doctest mostrando o comportamento de `averager`

```
>>> coro_avg = averager()
>>> next(coro_avg)
>>> coro_avg.send(10) ❶
```

```
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> coro_avg.send(None) ❷
Traceback (most recent call last):
...
StopIteration: Result(count=3, average=15.5)
```

❶ Essa versão não produz valores.

❷ Enviar `None` finaliza o laço, fazendo a corrotina terminar devolvendo o resultado. Como ocorre normalmente, o objeto gerador levanta `StopIteration`. O atributo `value` da exceção contém o valor devolvido.

Observe que o valor da expressão `return` é contrabandeado para o chamador na forma de um atributo da exceção `StopIteration`. É uma espécie de hack, mas preserva o comportamento existente dos objetos geradores: levantar `StopIteration` quando o gerador esgota.

O exemplo 16.15 mostra como recuperar o valor devolvido pela corrotina.

Exemplo 16.15 – Capturar `StopIteration` nos permite obter o valor devolvido por `averager`

```
>>> coro_avg = averager()
>>> next(coro_avg)
>>> coro_avg.send(10)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> try:
...     coro_avg.send(None)
... except StopIteration as exc:
...     result = exc.value
...
>>> result
Result(count=3, average=15.5)
```

Essa maneira sinuosa de obter o valor de retorno de uma corrotina faz mais sentido quando percebemos que ela foi definida como parte da PEP 380, e a construção `yield from` trata isso automaticamente capturando `StopIteration` internamente. É análogo ao uso de `StopIteration` em laços `for`: a exceção é tratada pelo mecanismo do laço de uma maneira transparente ao usuário. No caso de `yield from`, o interpretador não só

consume `StopIteration`, como seu atributo `value` torna-se o valor da própria expressão `yield from`. Infelizmente, não podemos testar isso interativamente no console, pois é um erro de sintaxe usar `yield from` – ou apenas `yield` – fora de uma função.⁴

A próxima seção tem um exemplo em que a corrotina `averager` é usada com `yield from` para produzir um resultado, de acordo com o propósito da PEP 380. Vamos então encarar `yield from`.

Usando `yield from`

A primeira coisa a saber sobre `yield from` é que é uma construção totalmente nova na linguagem. Ela faz muito mais que `yield`, tanto que é possível argumentar que a reutilização dessa palavra reservada causa confusão. Construções semelhantes em outras linguagens são chamadas de `await`, e esse é um nome muito melhor, pois transmite uma informação crucial: quando um gerador `gen` chama `yield from subgen()`, `subgen` assume o controle e produz valores a quem chamou `gen`; com efeito, o chamador acionará `subgen` diretamente. Enquanto isso, `gen` ficará bloqueado, esperando `subgen` terminar.⁵

No capítulo 14, vimos que `yield from` pode ser usado como um atalho para `yield` em um laço `for`. Por exemplo, isto:

```
>>> def gen():
...     for c in 'AB':
...         yield c
...     for i in range(1, 3):
...         yield i
...
>>> list(gen())
['A', 'B', 1, 2]
```

pode ser escrito como:

```
>>> def gen():
...     yield from 'AB'
...     yield from range(1, 3)
... 
```

4 Há uma extensão para iPython chamada `ipython-yf` (<https://github.com/teckil/ipython-yf>) que permite avaliar `yield from` diretamente no console de iPython. Ela é usada para testar código assíncrono e funciona com `asyncio`. Foi submetida como um patch para Python 3.5, mas não foi aceita. Consulte a *Issue #22412: Towards an asyncio-enabled command line* (Em direção a uma linha de comando que funcione com `asyncio`, <http://bugs.python.org/issue22412>) no sistema de monitoração de bugs de Python.

5 Quando escrevi este livro, havia uma PEP aberta propondo o acréscimo das palavras reservadas `await` e `async`: a PEP 492 — *Coroutines with `async` and `await` syntax* (Corrotinas com sintaxe `async` e `await`, <https://www.python.org/dev/peps/pep-0492/>).


```
>>> list(gen())  
['A', 'B', 1, 2]
```

Quando mencionamos `yield from` pela primeira vez na seção “Nova sintaxe em Python 3.3: `yield from`” na página 485, o código – reproduzido no exemplo 16.16 – mostrava um uso prático desse recurso.⁶

Exemplo 16.16 – Encadeando iteráveis com `yield from`

```
>>> def chain(*iterables):  
...     for it in iterables:  
...         yield from it  
...  
>>> s = 'ABC'  
>>> t = tuple(range(3))  
>>> list(chain(s, t))  
['A', 'B', 'C', 0, 1, 2]
```

Um exemplo um pouco mais complicado – porém mais útil – de `yield from` está em “Recipe 4.14. Flattening a Nested Sequence” (Linearizar uma sequência aninhada) do livro *Python Cookbook*, 3E (<http://shop.oreilly.com/product/0636920027072.do>)⁷, de Beazley e Jones (código-fonte disponível no GitHub em <http://bit.ly/1MMe1sc>).

A primeira coisa que a expressão `yield from x` faz com o objeto `x` é chamar `iter(x)` para obter um iterador a partir dele. Isso quer dizer que `x` pode ser qualquer iterável.

No entanto, se substituir laços `for` aninhados que produzam valores fosse a única contribuição de `yield from`, esse acréscimo na linguagem não teria uma boa chance de ser aceito. A verdadeira natureza de `yield from` não pode ser demonstrada com iteráveis simples; exige o uso de geradores aninhados – um verdadeiro desafio mental. É por isso que a PEP 380, que introduziu `yield from`, chama-se “Syntax for Delegating to a Subgenerator” (Sintaxe para delegar a um subgerador).

A principal funcionalidade de `yield from` é abrir um canal bidirecional do chamador mais externo para o subgerador mais interno, de modo que valores possam ser enviados e produzidos em ambas direções diretamente, e exceções possam ser lançadas até o subgerador mais interno, sem a adição de muito código repetitivo de tratamento de exceção nas corrotinas intermediárias. É isso que permite a delegação de corrotinas de uma maneira que não era possível antes.

O uso de `yield from` exige uma organização não trivial do código. Para falar dos componentes necessários, a PEP 380 usa alguns termos de maneira bem específica:

6 O exemplo 16.16 é apenas um exemplo didático. O módulo `itertools` já oferece uma função `chain` otimizada implementada em C.

7 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

gerador delegante (delegating generator)

A função geradora que contém a expressão `yield from <iterable>`.

subgerador (subgenerator)

O gerador obtido da parte `<iterable>` da expressão `yield from`. Esse é o “subgerador” mencionado no título da PEP 380: “Syntax for Delegating to a Subgenerator” (Sintaxe para delegar a um subgerador).

chamador (caller)

A PEP 380 usa o termo “chamador” para referir-se ao código do cliente que chama o gerador delegante. De acordo com o contexto, uso “cliente” em vez de “chamador” para distinguir do gerador delegante, que também é um “chamador” (ele chama o subgerador).



Com frequência, a PEP 380 usa a palavra “iterador” (iterator) para referir-se ao subgerador. Isso é confuso porque o gerador delegante também é um iterador. Sendo assim, prefiro usar o termo subgerador, de acordo com o título da PEP – “Syntax for Delegating to a Subgenerator” (Sintaxe para delegar a um subgerador). Entretanto o subgerador pode ser um iterador simples que implemente somente `__next__`, e `yield from` é capaz de tratar isso também, embora tenha sido criado para aceitar geradores que implementem `__next__`, `send`, `close` e `throw`.

O exemplo 16.17 oferece um contexto adicional para ver `yield from` em funcionamento, e a figura 16.2 identifica as partes relevantes do exemplo.⁸

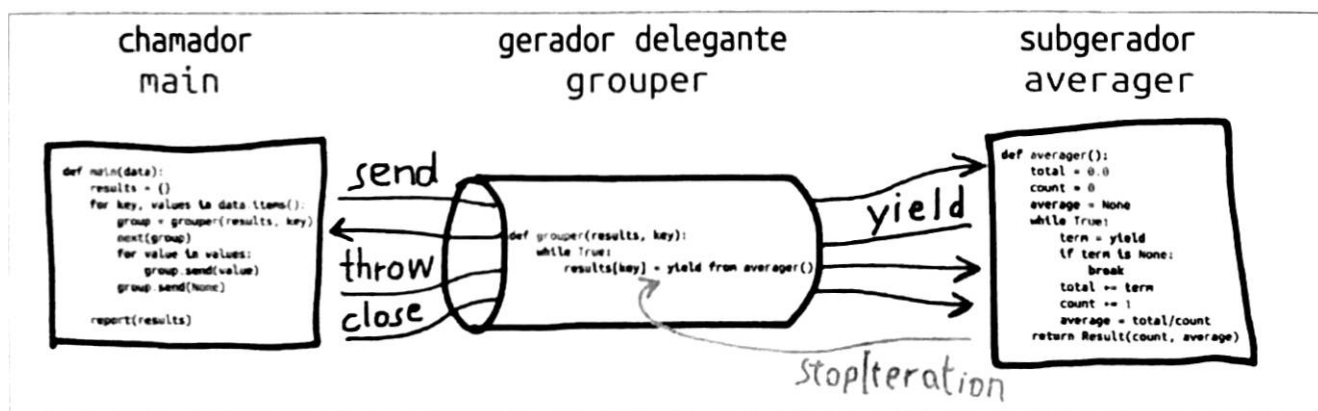


Figura 16.2 – Enquanto o gerador delegante está suspenso em `yield from`, o chamador envia dados diretamente ao subgerador, que entrega dados de volta ao chamador. O gerador delegante retoma a execução quando o subgerador retorna e o interpretador levanta `StopIteration` com o valor devolvido associado.

⁸ A figura 16.2 foi inspirada em um diagrama de Paul Sokolovsky (<http://flupy.org/resources/yield-from.pdf>).

O script *coroaverager3.py* lê um dict com pesos e alturas de meninas e meninos de uma classe fictícia de sétimo ano. Por exemplo, a chave 'boys;m' é mapeada para as alturas de nove meninos, em metros; 'girls;kg' são os pesos de dez meninas em quilogramas. O script alimenta a corrotina *averager*, que vimos antes, com os dados de cada grupo, e produz um relatório como este:

```
$ python3 coroaverager3.py
9 boys averaging 40.42kg
9 boys averaging 1.39m
10 girls averaging 42.04kg
10 girls averaging 1.43m
```

O código do exemplo 16.17 certamente não é a solução mais simples para o problema, mas serve para mostrar *yield from* em ação. Esse exemplo foi inspirado em um exemplo dado em *What's New in Python 3.3* (O que há de novo em Python 3.3, <http://bit.ly/1HGmVq>).

Exemplo 16.17 – *coroaverager3.py*: usando *yield from* para acionar *averager* e informar estatísticas

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')

# o subgerador
def averager(): ❶
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield ❷
        if term is None: ❸
            break
        total += term
        count += 1
        average = total/count
    return Result(count, average) ❹

# o gerador delegante
def grouper(results, key): ❺
    while True: ❻
        results[key] = yield from averager() ❼

# o código do cliente, também conhecido como chamador
def main(data): ❽
    results = {}
```

```

for key, values in data.items():
    group = grouper(results, key) ❶
    next(group) ❷
    for value in values:
        group.send(value) ❸
    group.send(None) # importante! ❹

# print(results) # remova o comentário para depurar
report(results)

# exibe o relatório
def report(results):
    for key, result in sorted(results.items()):
        group, unit = key.split(';')
        print('{:2} {:5} averaging {:.2f}{}'.format(
            result.count, group, result.average, unit))

data = {
    'girls;kg':
        [40.9, 38.5, 44.3, 42.2, 45.2, 41.7, 44.5, 38.0, 40.6, 44.5],
    'girls;m':
        [1.6, 1.51, 1.4, 1.3, 1.41, 1.39, 1.33, 1.46, 1.45, 1.43],
    'boys;kg':
        [39.0, 40.8, 43.2, 40.8, 43.1, 38.6, 41.4, 40.6, 36.3],
    'boys;m':
        [1.38, 1.5, 1.32, 1.25, 1.37, 1.48, 1.25, 1.49, 1.46],
}

if __name__ == '__main__':
    main(data)

```

- ❶ Mesma corrotina `averager` do exemplo 16.13. Aqui, é o subgerador.
- ❷ Cada valor enviado pelo código do cliente em `main` será associado a `term` aqui.
- ❸ A condição crucial de término. Sem ela, um `yield from` que chamasse essa corrotina ficaria bloqueado para sempre.
- ❹ O `Result` devolvido será o valor da expressão `yield from` em `grouper`.
- ❺ `grouper` é o gerador delegante.
- ❻ Cada iteração nesse laço cria uma nova instância de `averager`; cada uma é um objeto gerador funcionando como uma corrotina.

- ⑦ Sempre que grouper recebe um valor, este é enviado à instância de `averager` por `yield from`; grouper ficará suspensa aqui enquanto a instância de `averager` estiver consumindo valores enviados pelo cliente. Quando uma instância de `averager` executar até o fim, o valor que ela devolve será associado a `results[key]`. O laço `while` então continua e cria outra instância de `averager` para consumir mais valores.
- ⑧ `main` é o código do cliente, ou “chamador” (caller), no jargão da PEP 380. Essa é a função que controla tudo.
- ⑨ `group` é um objeto gerador resultante da chamada a `grouper` com o dict `results` para coletar os resultados e uma `key` em particular. Funcionará como uma corrotina.
- ⑩ Prepara a corrotina.
- ⑪ Envia cada `value` para `grouper`. Esse valor acaba sendo enviado para a linha `term = yield` de `averager`; `grouper` não tem chance de vê-lo.
- ⑫ Envia `None` para `grouper`. Isso faz a instância atual de `averager` terminar e permite que `grouper` volte a executar, criando outro `averager` para o próximo grupo de valores.

A última linha numerada no exemplo 16.17 com o comentário “importante!” enfatiza uma linha de código crucial: `group.send(None)`, que termina um `averager` e inicia o próximo. Se você comentar essa linha, o script não produzirá nenhuma saída. Descomentar a linha `print(results)` próximo ao final de `main` revelará que o dict `results` está vazio no final.



Se quiser descobrir por conta própria por que nenhum resultado é coletado, será uma ótima maneira de exercitar sua compreensão de como `yield from` funciona. O código de `coroaverager3.py` está no repositório de código de *Python fluente* (<http://bit.ly/1JlofLL>). A explicação será dada a seguir.

Apresentamos a seguir uma visão geral do funcionamento do exemplo 16.17, explicando o que aconteceria se omitíssemos a chamada a `group.send(None)` marcada como “importante!” em `main`:

- Cada iteração do laço `for` mais externo cria uma nova instância de `grouper` chamada `group`; esse é o gerador delegante.
- A chamada a `next(group)` prepara o gerador `grouper` responsável pela delegação; esse gerador entra em seu laço `while True` e fica suspenso em `yield from`, após chamar o subgerador `averager`.
- O laço `for` interno chama `group.send(value)`; ele alimenta o subgerador `averager` diretamente. Enquanto isso, a instância atual `group` de `grouper` fica suspensa no `yield from`.

- Quando o laço `for` interno termina, a instância `group` ainda está suspensa no `yield from`, portanto a atribuição a `results[key]` no corpo de `grouper` ainda não aconteceu.
- Sem o último `group.send(None)` no laço `for` externo, o subgerador `averager` não termina, o gerador `group` que delega jamais é reativado e a atribuição a `results[key]` não acontece.
- Quando a execução volta ao início do laço `for` externo, uma nova instância de `grouper` é criada e associada a `group`. A instância anterior de `grouper` é removida pelo coletor de lixo (com sua própria instância do subgerador `averager` suspensa, sem terminar).



A principal lição desse experimento é: se um subgerador não terminar, o gerador delegante ficará suspenso para sempre em `yield from`. Isso não impedirá seu programa de prosseguir, pois `yield from` (assim como um `yield` simples) transfere o controle para o código do cliente (ou seja, a quem chamou o gerador delegante). Mas isso significa que haverá tarefas não finalizadas.

O exemplo 16.17 mostra a organização mais simples possível de `yield from`, com apenas um gerador delegante e um subgerador. Como o gerador delegante funciona como um cano (pipe), você pode conectar qualquer quantidade de geradores formando um encanamento (pipeline): um gerador delegante usa `yield from` para chamar um subgerador que, por sua vez, é um gerador delegante; ele chama outro subgerador com `yield from`, e assim sucessivamente. Em algum momento, esse encanamento precisa terminar em um gerador simples que use apenas `yield`, mas pode terminar também em qualquer objeto iterável, como no exemplo 16.16.

Todo encanamento com `yield from` precisa ser acionado por um cliente que chama `next(...)` ou `.send(...)` no gerador delegante mais externo. Essa chamada pode ser implícita, por exemplo, um laço `for`.

Vamos agora analisar a descrição formal da construção `yield from`, conforme apresentada na PEP 380.

O significado de `yield from`

Ao desenvolver a PEP 380, Greg Ewing – seu autor – foi questionado sobre a complexidade da semântica proposta. Uma de suas respostas foi: “Para os seres humanos, quase todas as informações importantes estão contidas em um parágrafo próximo ao início.” Ele então citou parte do rascunho da PEP 380 que, naquela época, continha o seguinte:

“Quando o iterador é outro gerador, o efeito é o mesmo de ter o corpo do subgerador incluído (inline) no ponto em que está a expressão `yield from`. Além do mais, o subgerador pode executar um comando `return` com um valor, e esse valor será o valor da expressão `yield from`.⁹

Essas palavras tranquilizadoras não fazem mais parte da PEP – porque não cobrem todos os casos bizarros. Mas são boas como uma primeira aproximação.

A versão aprovada da PEP 380 explica o comportamento de `yield from` em seis itens na seção Proposal (Proposta, <https://www.python.org/dev/peps/pep-0380/#proposal>). Reproduzi-os quase literalmente aqui, embora tenha substituído toda ocorrência da palavra ambígua “iterador” por “subgerador” e acrescentado alguns esclarecimentos. O exemplo 16.17 ilustra os quatro pontos a seguir:

- Qualquer valor que o subgerador produz é passado diretamente a quem chama o gerador delegante (ou seja, o código do cliente).
- Qualquer valor enviado ao gerador delegante usando `send()` é passado diretamente ao subgerador. Se o valor enviado for `None`, o método `__next__()` do subgerador será chamado. Se o valor enviado não for `None`, o método `send()` do subgerador será chamado. Se a chamada levantar `StopIteration`, o gerador delegante retomará a execução. Qualquer outra exceção será propagada até o gerador delegante.
- `return expr` em um gerador (ou subgerador) faz `StopIteration(expr)` ser levantada na saída do gerador.
- O valor da expressão `yield from` é o primeiro argumento da exceção `StopIteration` levantada pelo subgerador quando terminar.

As outras duas características de `yield from` têm a ver com exceções e término:

- As exceções que não sejam `GeneratorExit`, lançadas pelo chamador no gerador delegante usando `.throw()`, são passadas ao método `.throw()` do subgerador. Se a chamada levantar `StopIteration`, o gerador delegante retomará a execução. Qualquer outra exceção será propagada até o gerador delegante.
- Se uma exceção `GeneratorExit` for lançada no gerador delegante via `.throw()` ou o método `.close()` do gerador delegante for chamado, o método `.close()` do subgerador será chamado, se existir. Se essa chamada resultar em uma exceção, ela será propagada até o gerador delegante. Caso contrário, `GeneratorExit` será levantada no gerador delegante.

A semântica detalhada de `yield from` é sutil, especialmente os pontos que lidam com exceções. Greg Ewing fez um ótimo trabalho colocando-a em palavras na PEP 380.

⁹ Mensagem para Python-Dev: “PEP 380 (yield from a subgenerator) comments” [Comentários para a PEP 380 (yield from com um subgerador), <http://bit.ly/1JlopTu>, 21 de março de 2009].

Ewing também documentou o comportamento de `yield from` usando pseudocódigo (com sintaxe Python). Pessoalmente, acho útil investir tempo estudando o pseudocódigo da PEP 380. No entanto o pseudocódigo tem 40 linhas, e não é fácil compreendê-lo à primeira vista.

Uma boa maneira de abordar esse pseudocódigo é simplificá-lo para que trate apenas o caso de uso mais básico e comum de `yield from`.

Considere que `yield from` apareça em um gerador delegante. O código do cliente aciona o gerador delegante, que aciona o subgerador. Então, para simplificar a lógica envolvida, vamos supor que o cliente jamais chame `.throw(...)` ou `.close()` no gerador delegante. Vamos também supor que o subgerador jamais levanta uma exceção até terminar, quando `StopIteration` é levantada pelo interpretador.

O exemplo 16.17 é um script em que essas suposições para simplificação são válidas. De fato, em muitos códigos do mundo real, espera-se que o gerador delegante execute até terminar. Portanto vamos ver como `yield from` funciona nesse mundo mais feliz e mais simples.

Dê uma olhada no exemplo 16.18, que é uma expansão desta simples instrução a seguir, no corpo do gerador delegante:

```
RESULT = yield from EXPR
```

Procure seguir a lógica do exemplo 16.18.

Exemplo 16.18 – Pseudocódigo simplificado, equivalente à instrução `RESULT = yield from EXPR` no gerador delegante (aborda o caso mais simples: sem suporte a `.throw(...)` e a `.close()`; a única exceção tratada é `StopIteration`)

```
_i = iter(EXPR) ❶
try:
    _y = next(_i) ❷
except StopIteration as _e:
    _r = _e.value ❸
else:
    while 1: ❹
        _s = yield _y ❺
        try:
            _y = _i.send(_s) ❻
        except StopIteration as _e: ❼
            _r = _e.value
            break
RESULT = _r ❽
```

- ❶ `EXPR` pode ser qualquer iterável, pois `iter()` é aplicado para obter um iterador `_i` (esse é o subgerador).

- ❶ O subgerador é preparado; o resultado é armazenado para ser o primeiro valor `_y` produzido.
- ❷ Se `StopIteration` for levantada, extrairá o atributo `value` da exceção e o atribuirá a `_r`: esse é o `RESULT` no caso mais simples.
- ❸ Enquanto esse laço está executando, o gerador delegante fica bloqueado, funcionando apenas como um canal entre o chamador e o subgerador.
- ❹ Produz o item mais recente produzido pelo subgerador; espera um valor `_s` enviado pelo chamador. Observe que esse é o único `yield` nessa listagem.
- ❺ Tenta avançar o subgerador, encaminhando o `_s` enviado pelo chamador.
- ❻ Se o subgerador tiver levantado `StopIteration`, obterá `value`, o atribuirá a `_r` e sairá do laço, retomando a execução do gerador delegante.
- ❼ `_r` é o `RESULT`: o valor de toda a expressão `yield from`.

Nesse pseudocódigo simplificado, preservei os nomes das variáveis usadas no pseudocódigo publicado na PEP 380. As variáveis são:

`_i` (*iterator*)

O subgerador

`_y` (*yielded*)

Um valor produzido pelo subgerador

`_r` (*result*)

O resultado final (ou seja, o valor da expressão `yield from` quando o subgerador termina)

`_s` (*sent*)

Um valor enviado pelo chamador ao gerador delegante, encaminhado ao subgerador

`_e` (*exception*)

Uma exceção (é sempre uma instância de `StopIteration` nesse pseudocódigo simplificado)

Além de não tratar `.throw(...)` nem `.close()`, o pseudocódigo simplificado sempre usa `.send(...)` para encaminhar ao subgerador as chamadas a `next()` ou `.send(...)` feitas pelo cliente. Não se preocupe com essas distinções detalhadas em uma leitura inicial. Como mencionamos, o exemplo 16.17 executaria perfeitamente se `yield from` fizesse somente o que está no pseudocódigo simplificado do exemplo 16.18.

Mas a realidade é mais complicada por causa da necessidade de tratar chamadas a `.throw(...)` e `.close()` feitas pelo cliente, que devem ser passadas ao subgerador. Além disso, o subgerador pode ser um iterador simples, que não aceita `.throw(...)` ou `.close()`, portanto eles devem ser tratados pela lógica de `yield from`. Se o subgerador implementar esses métodos, ambos farão exceções serem levantadas no subgerador, as quais devem também ser tratadas pelo aparato de `yield from`. O subgerador também pode lançar exceções próprias, não provocadas pelo chamador, e isso deve ser igualmente tratado pela implementação de `yield from`. Por fim, como otimização, se o código do cliente chamar `next(...)` ou `.send(None)`, ambos serão encaminhados como uma chamada a `next(...)` no subgerador; somente se o chamador enviar um valor diferente de `None` é que o método `.send(...)` do subgerador será usado.

Para sua conveniência, a seguir está o pseudocódigo completo da expansão de `yield from` da PEP 380, com anotações. O exemplo 16.19 foi literalmente copiado; acrescentei somente os números para os comentários.

Novamente, o código mostrado no exemplo 16.19 é uma expansão desta instrução única no corpo do gerador delegante:

```
RESULT = yield from EXPR
```

Exemplo 16.19 – Pseudocódigo equivalente à instrução `RESULT = yield from EXPR` no gerador delegante

```
_i = iter(EXPR) ❶
try:
    _y = next(_i) ❷
except StopIteration as _e:
    _r = _e.value ❸
else:
    while 1: ❹
        try:
            _s = yield _y ❺
        except GeneratorExit as _e: ❻
            try:
                _m = _i.close
            except AttributeError:
                pass
            else:
                _m()
            raise _e
        except BaseException as _e: ❼
            _x = sys.exc_info()
            try:
                _m = _i.throw
```

```

    except AttributeError:
        raise _e
    else: ❸
        try:
            _y = _m(*_x)
        except StopIteration as _e:
            _r = _e.value
            break
    else: ❹
        try: ❺
            if _s is None: ❻
                _y = next(_i)
            else:
                _y = _i.send(_s)
        except StopIteration as _e: ❼
            _r = _e.value
            break

RESULT = _r ❺

```

- ❶ `EXPR` pode ser qualquer iterável, pois `iter()` é aplicado para obter um iterador `_i` (esse é o subgerador).
- ❷ O subgerador é preparado; o resultado é armazenado para ser o primeiro valor `_y` produzido.
- ❸ Se `StopIteration` for levantada, extrairá o atributo `value` da exceção e o atribuirá a `_r`: esse é o `RESULT` no caso mais simples.
- ❹ Enquanto esse laço está executando, o gerador delegante fica bloqueado, funcionando apenas como um canal entre o chamador e o subgerador.
- ❺ Entrega o item atual produzido pelo subgerador; espera um valor `_s` ser enviado pelo chamador. Esse é o único `yield` nessa listagem.
- ❻ Esse código lida com o encerramento do gerador delegante e do subgerador. Como o subgerador pode ser qualquer iterador, talvez ele não tenha um método `close`.
- ❼ Esse código lida com exceções lançadas pelo chamador usando `.throw()`. Novamente, o subgerador pode ser um iterador sem um método `throw` para ser chamado – caso em que a exceção é levantada no gerador delegante.
- ❽ Se o subgerador tem um método `throw`, chama-o com a exceção passada pelo chamador. O subgerador pode tratar a exceção (e o laço continua), pode levantar `StopIteration` (o resultado `_r` é extraído dela e o laço termina) ou pode levantar a mesma exceção ou outra, que não é tratada aqui e se propaga até o gerador delegante.

- ⑨ Se nenhuma exceção for recebida na produção do valor...
- ⑩ Tenta avançar o subgerador...
- ⑪ Chama `next` no subgerador se o último valor recebido do chamador for `None`; caso contrário, chama `send`.
- ⑫ Se o subgerador tiver levantado `StopIteration`, obterá `value`, o atribuirá a `_r` e sairá do laço, retomando a execução do gerador delegante.
- ⑬ `_r` é o `RESULT`: o valor de toda a expressão `yield from`.

A maior parte da lógica do pseudocódigo de `yield from` está implementada em seis blocos `try/except` aninhados com até quatro níveis de profundidade, portanto é um pouco difícil de ler. Os únicos comandos de controle de fluxo além dos `try/except` são um `while`, um `if` e um `yield`. Procure no código onde estão `while`, `yield` e as chamadas a `next(...)` e `.send(...)`: elas ajudarão você a ter uma ideia de como a estrutura como um `todo` funciona.

Bem no início do exemplo 16.19, um detalhe importante revelado pelo pseudocódigo é o fato de o subgerador ser preparado (linha com o número 2 no exemplo 16.19).¹⁰ Isso quer dizer que decoradores que fazem a preparação automaticamente, como aqueles da seção “Decoradores para preparação de corrotinas” na página 522, são incompatíveis com `yield from`.

Na mesma mensagem (<http://bit.ly/1JlopTu>) da qual extraí a citação no início desta seção, Greg Ewing diz o seguinte sobre a expansão do pseudocódigo de `yield from`:

Não temos a intenção de que você aprenda isso lendo a expansão – ela está presente apenas para mostrar todos os detalhes aos advogados da linguagem¹¹.

Focar nos detalhes da expansão do pseudocódigo pode não ser útil – dependendo de seu estilo de aprendizado. Estudar um código de verdade que use `yield from` certamente vale mais a pena que ficar quebrando a cabeça com o pseudocódigo de sua implementação. Entretanto quase todos os exemplos com `yield from` que vi estão associados à programação assíncrona com o módulo `asyncio`, portanto dependem de um laço de eventos ativo para funcionar. Veremos `yield from` várias vezes no capítulo 18. Há alguns links para códigos interessantes que usam `yield from` sem um laço de eventos na seção “Leituras complementares” na página 555.

Vamos agora passar para um exemplo clássico de uso de corrotina: programar simulações. Esse exemplo não é uma aplicação de `yield from`, mas mostra como as corrotinas são usadas para administrar atividades concorrentes em uma única `thread`.

¹⁰ Em uma mensagem para Python-ideas em 5 de abril de 2009 (<http://bit.ly/1JloXJ1>), Nick Coghlan questionou se a preparação implícita feita por `yield from` era uma boa ideia.

¹¹ Nota do autor/revisor: “Advogados da linguagem” é uma tradução literal de “language lawyers”, um termo usado na documentação de Python para se referir àqueles que discutem os detalhes mais minuciosos da linguagem.

Caso de uso: corrotinas para uma simulação de eventos discretos

Corrotinas são uma maneira natural de expressar muitos algoritmos como simulações, jogos, E/S assíncrona e outras formas de programação orientada a eventos ou multitarefa cooperativa.¹²

— Guido van Rossum e Phillip J. Eby

PEP 342 – Coroutines via Enhanced Generators (Corrotinas por meio de geradores melhorados)

Nesta seção, descreverei uma simulação bem simples, implementada somente com corrotinas e objetos da biblioteca-padrão. A simulação é uma aplicação clássica para corrotinas na literatura da ciência da computação. Simula, a primeira linguagem orientada a objetos, introduziu o conceito de corrotinas exatamente para dar suporte a simulações.



A motivação para o exemplo de simulação a seguir não é acadêmica. Corrotinas são o bloco de construção fundamental do pacote `asyncio`. Uma simulação mostra como implementar atividades concorrentes usando corrotinas em vez de threads – e isso ajudará bastante quando discutirmos o pacote `asyncio` no capítulo 18.

Antes de iniciar o exemplo, algumas palavras sobre simulações.

Sobre simulações de eventos discretos

Uma SED (Simulação de Eventos Discretos ou Discrete Event Simulation) é um tipo de simulação em que um sistema é modelado como uma sequência de eventos. Em uma SED, o “relógio” da simulação não avança em incrementos fixos, mas pula diretamente para o instante simulado do próximo evento modelado. Por exemplo, se estivermos simulando a operação de um táxi em alto nível, um evento será pegar um passageiro e o próximo será deixá-lo. Não importa se o percurso durou 5 ou 50 minutos: quando o evento de deixar o passageiro ocorrer, o relógio será atualizado com o instante final da corrida em uma única operação. Em uma SED, podemos simular um ano de corridas de táxi em menos de um segundo. Isso é o oposto de uma simulação contínua, em que o relógio avança continuamente, sendo incrementado com um valor fixo – e geralmente pequeno.

Intuitivamente, jogos baseados em rodadas são exemplos de simulações de eventos discretos: o estado do jogo só muda quando um jogador faz um movimento e, enquanto um jogador está decidindo qual será o próximo movimento, o relógio da simulação fica congelado. Jogos em tempo real, por outro lado, são simulações contínuas, em que o relógio da simulação funciona o tempo todo, o estado do jogo é atualizado várias vezes por segundo e jogadores mais lentos ficam em desvantagem.

¹² Frase de início da seção “Motivation” (Motivação) da PEP 342 (<https://www.python.org/dev/peps/pep-0342/>).

Os dois tipos de simulações podem ser escritos com várias threads ou com uma única thread usando técnicas de programação orientada a eventos, como callbacks ou corrotinas acionadas por um laço de eventos. Pode-se argumentar que é mais natural implementar uma simulação contínua usando threads para dar conta das ações que ocorrem em paralelo em tempo real. Por outro lado, as corrotinas oferecem exatamente a abstração correta para implementar uma SED. SimPy¹³ é um pacote Python para SED, que usa uma corrotina para representar cada processo na simulação.



Na área de simulação, o termo *processo* refere-se às atividades de uma entidade do modelo, e não a um processo do sistema operacional. Um processo da simulação pode ser implementado como um processo do sistema operacional, mas, normalmente, uma thread ou uma corrotina é usada para isso.

Se estiver interessado em simulações, vale a pena estudar o SimPy. Entretanto, nesta seção, descreverei uma SED bem simples, implementada apenas com recursos da biblioteca-padrão. Meu objetivo é ajudar você a desenvolver uma intuição sobre programação de ações concorrentes usando corrotinas. Entender a próxima seção exigirá um estudo cuidadoso, mas a recompensa virá na forma de esclarecimentos sobre como bibliotecas como *asyncio*, *Twisted* e *Tornado* são capazes de administrar muitas atividades concorrentes usando uma única thread de execução. .

A simulação da frota de táxis

Em nosso programa de simulação *taxi_sim.py*, vários táxis são criados. Cada um fará um número fixo de corridas e então irá para casa. Um táxi sai da garagem e começa a circular – procurando um passageiro. Isso dura até um passageiro pegar o táxi e o percurso começar. Quando o passageiro é deixado, o táxi volta a circular.

O tempo transcorrido circulando e fazendo corridas é gerado por meio de distribuição exponencial. Para uma exibição mais limpa, os tempos estão em minutos inteiros, mas a simulação funcionaria também com intervalos em *float*.¹⁴ Cada mudança de estado de cada táxi é informada como um evento. A figura 16.3 mostra um exemplo de execução do programa.

13 Consulte a documentação oficial de SimPy (<http://bit.ly/IHG54Oz>) – não confunda com SymPy (<http://bit.ly/IHG53KI>), uma biblioteca bem conhecida para matemática simbólica, mas sem relação com a primeira.

14 Não sou especialista em operações de frota de táxis, portanto não leve meus números a sério. Distribuições exponenciais são comumente usadas em SED. Você verá algumas corridas muito rápidas. Simplesmente imagine que é um dia chuvoso e que alguns passageiros estão usando táxis somente para ir até a próxima esquina – em uma cidade ideal, onde há táxis quando chove.

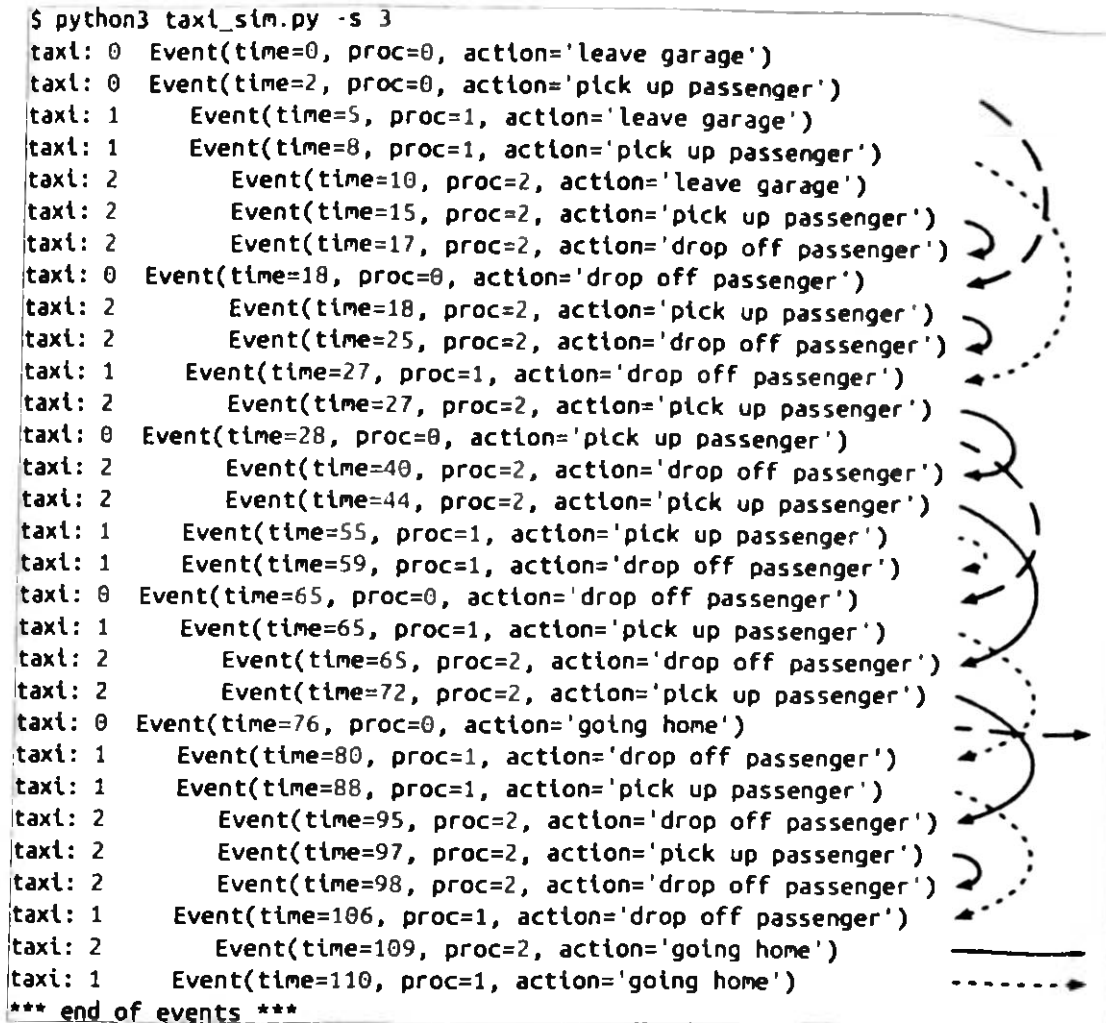


Figura 16.3 – Exemplo de execução de `taxi_sim.py` com três táxis. O argumento `-s 3` define a semente (seed) do gerador aleatório de modo que as execuções do programa possam ser reproduzidas para depuração e demonstração. As setas destacam as corridas dos táxis.

O aspecto mais importante a observar na figura 16.3 é o entrelaçamento entre as corridas dos três táxis. Acrescentei manualmente as setas para ver mais facilmente as corridas dos táxis: cada seta começa quando o táxi pega um passageiro e termina quando o passageiro é deixado. Intuitivamente, isso mostra como as corrotinas podem ser usadas para administrar atividades concorrentes.

Outros aspectos a serem observados na figura 16.3 são:

- Cada táxi deixa a garagem cinco minutos após o outro.
- Foram necessários dois minutos para o táxi 0 pegar o primeiro passageiro em `time=2`; três minutos para o táxi 1 (`time=8`) e cinco minutos para o táxi 2 (`time=15`).
- O motorista do táxi 0 faz apenas duas corridas (setas com tracejado longo): a primeira começa em `time=2` e termina em `time=18`; a segunda começa em `time=28` e termina em `time=65` – a corrida mais longa nessa execução da simulação.
- O táxi 1 faz quatro corridas (setas com tracejado curto) e então vai para casa em `time=110`.

- O táxi 2 faz seis corridas (setas contínuas) e então vai para casa em `time=109`. Sua última corrida dura apenas um minuto, começando em `time=97`.¹⁵
- Enquanto o táxi 1 está fazendo sua primeira corrida, começando em `time=8`, o táxi 2 deixa a garagem em `time=10` e completa duas corridas (setas contínuas menores).
- Nesse exemplo de execução, todos os eventos agendados foram concluídos no tempo default de simulação de 180 minutos; o último evento ocorreu em `time=110`.

A simulação também pode terminar com eventos pendentes. Quando isso acontece, a mensagem final informará algo como:

```
*** end of simulation time: 3 events pending ***
```

A listagem completa de *taxi_sim.py* está no exemplo A.6. Neste capítulo, mostraremos apenas as partes relevantes ao nosso estudo de corrotinas. As funções realmente importantes são apenas duas: `taxi_process` (uma corrotina) e o método `Simulator.run`, em que o laço principal da simulação é executado.

O exemplo 16.20 mostra o código de `taxi_process`. Essa corrotina usa dois objetos definidos em outro lugar: a função `compute_delay`, que devolve um intervalo de tempo em minutos, e a classe `Event`, que é uma `namedtuple` definida como:

```
Event = collections.namedtuple('Event', 'time proc action')
```

Em uma instância de `Event`, `time` é o instante em que o evento ocorrerá na simulação, `proc` é o identificador da instância do processo associado ao táxi e `action` é uma string que descreve a atividade.

Vamos analisar `taxi_process` passo a passo no exemplo 16.20.

Exemplo 16.20 – *taxi_sim.py*: corrotina `taxi_process` que implementa as atividades de cada táxi

```
def taxi_process(ident, trips, start_time=0): ❶
    """Cede o controle ao simulador gerando um evento a cada mudança de estado"""
    time = yield Event(start_time, ident, 'leave garage') ❷
    for i in range(trips): ❸
        time = yield Event(time, ident, 'pick up passenger') ❹
        time = yield Event(time, ident, 'drop off passenger') ❺
    yield Event(time, ident, 'going home') ❻
    # fim do processo associado ao táxi ❼
```

- ❶ `taxi_process` será chamado uma vez por táxi, criando um objeto gerador para representar suas operações. `ident` é o número do táxi (por exemplo, 0, 1, 2 na execução de exemplo); `trips` é o número de corridas que esse táxi fará antes de ir para casa; `start_time` é o instante em que o táxi deixa a garagem.

¹⁵ Eu era o passageiro. Percebi que tinha esquecido minha carteira.

- ❶ O primeiro Event produzido é 'leave garage'. Isso suspende a corrotina e deixa o laço principal da simulação continuar para o próximo evento agendado. Quando chegar o momento de reativar esse processo, o laço principal enviará (com `send`) o instante atual da simulação, que será atribuído a `time`.
- ❷ Esse bloco será repetido uma vez para cada corrida.
- ❸ Um Event sinalizando que o passageiro pegou o táxi é produzido. A corrotina faz uma pausa aqui. Quando chegar o momento de reativar essa corrotina, o laço principal enviará o instante atual (com `send`) novamente.
- ❹ Um Event sinalizando que o passageiro desceu do táxi é produzido. A corrotina é suspensa novamente, esperando o laço principal enviar-lhe o instante de sua reativação.
- ❺ O laço `for` termina após o número especificado de corridas e um evento 'going home' final é produzido. A corrotina será suspensa pela última vez. Ao ser reativada, ela receberá o instante do laço principal da simulação, mas, nesse local, não atribuirei esse valor a nenhuma variável, pois ele não será usado.
- ❻ Quando a corrotina chegar ao final, o objeto gerador levantará `StopIteration`.

Você mesmo pode “conduzir” (drive) um táxi chamando `taxi_process` no console de Python.¹⁶ O exemplo 16.21 mostra como isso é feito.

Exemplo 16.21 – Acionando a corrotina `taxi_process`

```
>>> from taxi_sim import taxi_process
>>> taxi = taxi_process(ident=13, trips=2, start_time=0) ❶
>>> next(taxi) ❷
Event(time=0, proc=13, action='leave garage')
>>> taxi.send(_.time + 7) ❸
Event(time=7, proc=13, action='pick up passenger') ❹
>>> taxi.send(_.time + 23) ❺
Event(time=30, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 5) ❻
Event(time=35, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 48) ❼
Event(time=83, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 1)
Event(time=84, proc=13, action='going home') ❽
>>> taxi.send(_.time + 10) ❾
```

¹⁶ O verbo “drive” (acionar, conduzir) é comumente usado para descrever a operação de uma corrotina: o código do cliente aciona a corrotina enviando-lhe valores. No exemplo 16.21, o código do cliente é o que você digita no console.

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

- ❶ Cria um objeto gerador para representar um táxi com `ident=13`; esse táxi fará duas corridas e começará a operar em `t=0`.
- ❷ Prepara a corrotina; produz o primeiro evento.
- ❸ Podemos agora enviar-lhe o instante atual (com `send`). No console, a variável `_` é associada ao último resultado; nesse caso, somei 7 ao tempo, o que significa que o táxi gastará 7 minutos procurando o primeiro passageiro.
- ❹ Isso é produzido pelo laço `for` no início da primeira corrida.
- ❺ Enviar `_.time + 23` significa que a corrida com o primeiro passageiro terá uma duração de 23 minutos.
- ❻ Então o táxi circulará por 5 minutos.
- ❼ A última corrida durará 48 minutos.
- ❽ Após duas corridas completas, o laço termina e o evento 'going home' é produzido.
- ❾ A próxima tentativa de enviar dados (com `send`) à corrotina a faz avançar para o final. Em seu retorno, o interpretador levanta `StopIteration`.

Observe que, no exemplo 16.21, usei o console para emular o laço principal da simulação. Obtive o atributo `.time` de um `Event` produzido pela corrotina `taxi`, somei um número arbitrário e usei a soma na próxima chamada a `taxi.send` para reativá-la. Na simulação, as corrotinas dos táxis são acionadas pelo laço principal no método `Simulator.run`. O “relógio” da simulação é mantido na variável `sim_time` e é atualizado com o instante de cada evento produzido.

Para instanciar a classe `Simulator`, a função `main` de `taxi_sim.py` cria um dicionário `taxis` desta maneira:

```
taxis = {i: taxi_process(i, (i + 1) * 2, i * DEPARTURE_INTERVAL)
         for i in range(num_taxis)}
sim = Simulator(taxis)
```

`DEPARTURE_INTERVAL` é 5; se `num_taxis` for 3, como no exemplo da execução, as linhas anteriores farão o mesmo que:

```
taxis = {0: taxi_process(ident=0, trips=2, start_time=0),
         1: taxi_process(ident=1, trips=4, start_time=5),
         2: taxi_process(ident=2, trips=6, start_time=10)}
sim = Simulator(taxis)
```

Desse modo, os valores do dicionário `taxis` serão três objetos geradores distintos, com parâmetros diferentes. Por exemplo, o táxi 1 fará quatro corridas e começará a procurar passageiros em `start_time=5`. Esse dict é o único argumento necessário para criar uma instância de `Simulator`.

O exemplo 16.22 mostra o método `Simulator.__init__`. As principais estruturas de dados de `Simulator` são:

`self.events`

Uma `PriorityQueue` para armazenar instâncias de `Event`. Uma `PriorityQueue` permite inserir itens (com `put`) e então obtê-los (com `get`) de forma ordenada pelo `item[0]`; ou seja, pelo atributo `time` no caso de nossos objetos `Event` do tipo `namedtuple`.

`self.procs`

Um dict que mapeia cada número de processo a um processo ativo da simulação – um objeto gerador que representa um táxi. Será associado a uma cópia do dict `taxis` mostrado antes.

Exemplo 16.22 – `taxi_sim.py`: inicializador da classe `Simulator`

```
class Simulator:
```

```
    def __init__(self, procs_map):
        self.events = queue.PriorityQueue() ❶
        self.procs = dict(procs_map) ❷
```

- ❶ `PriorityQueue` para armazenar os eventos agendados, ordenados pelos instantes dos eventos em ordem crescente.
- ❷ Recebemos o argumento `procs_map` como um dict (ou qualquer mapeamento), mas criamos um dict a partir dele, para termos uma cópia local, porque quando a simulação executar, cada táxi que for para casa será removido de `self.procs`, e não queremos alterar o objeto passado pelo usuário.

Filas com prioridade são um bloco de construção fundamental em simulações de eventos discretos; os eventos são criados em qualquer ordem, colocados na fila e, posteriormente, são recuperados em ordem, de acordo com o instante agendado para cada um. Por exemplo, os dois primeiros eventos colocados na fila poderiam ser:

```
Event(time=14, proc=0, action='pick up passenger')
Event(time=11, proc=1, action='pick up passenger')
```

Isso quer dizer que o táxi 0 levará 14 minutos para pegar o primeiro passageiro, enquanto o táxi 1 – começando em `time=10` – gastará um minuto e pegará um passageiro em `time=11`. Se esses dois eventos estiverem na fila, o primeiro evento que o laço principal obtiver da fila com prioridades será `Event(time=11, proc=1, action='pick up passenger')`.

Vamos agora estudar o algoritmo principal da simulação, ou seja, o método `Simulator.run`. Ele é chamado pela função `main` logo depois de `Simulator` ser instanciado, assim:

```
sim = Simulator(taxis)
sim.run(end_time)
```

A listagem da classe `Simulator` com os números para comentários está no exemplo 16.23, mas, a seguir, apresentamos uma visão geral do algoritmo implementado em `Simulator.run`:

1. Percorre os processos que representam táxis em um laço.
 - a. Prepara a corrotina para cada táxi chamando `next()`. Isso fará o primeiro `Event` ser produzido para cada táxi.
 - b. Coloca cada evento na fila `self.events` de `Simulator`.
2. Executa o laço principal da simulação enquanto `sim_time < end_time`.
 - a. Verifica se `self.events` está vazio; se estiver, sai do laço.
 - b. Obtém `current_event` de `self.events`. Este será o objeto `Event` com o menor `time` em `PriorityQueue`.
 - c. Exibe `Event`.
 - d. Atualiza o instante da simulação com o atributo `time` de `current_event`.
 - e. Envia o instante à corrotina identificada pelo atributo `proc` de `current_event`. A corrotina produzirá `next_event`.
 - f. Agenda `next_event` adicionando-o à fila `self.events`.

A classe `Simulator` completa está no exemplo 16.23.

Exemplo 16.23 – taxi_sim.py: `Simulator`, uma classe enxuta para simulação de eventos discretos; o foco está no método `run`

```
class Simulator:
```

```
    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)

    def run(self, end_time): ❶
        """Agenda e exibe eventos até o tempo acabar"""
        # agenda o primeiro evento para cada táxi
        for _, proc in sorted(self.procs.items()): ❷
            first_event = next(proc) ❸
            self.events.put(first_event) ❹
```

```

# laço principal da simulação
sim_time = 0 ❹
while sim_time < end_time: ❶
    if self.events.empty(): ❷
        print('*** end of events ***')
        break

    current_event = self.events.get() ❸
    sim_time, proc_id, previous_action = current_event ❹
    print('taxi:', proc_id, proc_id * ' ', current_event) ❺
    active_proc = self.procs[proc_id] ❻
    next_time = sim_time + compute_duration(previous_action) ❼
    try:
        next_event = active_proc.send(next_time) ❽
    except StopIteration:
        del self.procs[proc_id] ❿
    else:
        self.events.put(next_event) ⓫
else: ⓬
    msg = '*** end of simulation time: {} events pending ***'
    print(msg.format(self.events.qsize()))

```

- ❶ O `end_time` da simulação é o único argumento obrigatório para `run`.
- ❷ Usa `sorted` para recuperar itens de `self.procs` ordenados pela chave; a chave não nos interessa, portanto ela é atribuída a `_`.
- ❸ `next(proc)` prepara cada corrotina fazendo-a avançar até o primeiro `yield` para que esteja pronta para receber dados. Um `Event` é produzido.
- ❹ Adiciona cada evento à `PriorityQueue self.events`. O primeiro evento de cada táxi é `'leave garage'`, como vimos no exemplo de execução (Exemplo 16.20).
- ❺ Zera `sim_time`, que é o relógio da simulação.
- ❻ Laço principal da simulação: executa enquanto `sim_time` for menor que `end_time`.
- ❼ O laço principal também pode terminar se não houver eventos pendentes na fila.
- ❽ Obtém `Event` com o menor `time` da fila com prioridades; esse é o `current_event`.
- ❹ Desempacota os dados de `Event`. Essa linha atualiza o relógio da simulação, `sim_time`, para que reflita o instante em que o evento ocorreu.¹⁷
- ❿ Exibe `Event`, identificando o táxi e acrescentando indentação de acordo com o ID do táxi.

¹⁷ Isso é típico em uma simulação de eventos discretos: o relógio da simulação não é incrementado por um valor fixo a cada iteração do laço, mas avança de acordo com a duração de cada evento concluído.

- ⑪ Recupera a corrotina para o táxi ativo a partir do dicionário `self.procs`.
- ⑫ Calcula o próximo instante de ativação somando `sim_time` e o resultado da chamada a `compute_duration(...)` com a ação anterior (por exemplo, 'pick up passenger', 'drop off passenger' etc.).
- ⑬ Envia `time` para a corrotina do táxi. A corrotina produzirá `next_event` ou levantará `StopIteration` quando terminar.
- ⑭ Se `StopIteration` for levantada, apagará a corrotina do dicionário `self.procs`.
- ⑮ Caso contrário, inserirá `next_event` na fila.
- ⑯ Se o laço terminar porque a duração da simulação expirou, exibirá a quantidade de eventos pendentes (que, às vezes, pode ser 0 por coincidência).

Fazendo uma ligação com o capítulo 15, observe que o método `Simulator.run` no exemplo 16.23 usa blocos `else` em dois lugares que não são comandos `if`:

- O laço `while` principal tem um comando `else` para informar que a simulação terminou porque `end_time` foi alcançado – e não porque não havia mais eventos para processar.
- O comando `try` no final do laço `while` tenta obter um `next_event` enviando `next_time` para o processo de táxi atual e, se houver sucesso, o bloco `else` colocará `next_event` na fila `self.events`.

Acredito que o código de `Simulator.run` seria um pouco mais difícil de ler sem esses blocos `else`.

A questão principal nesse exemplo foi mostrar um laço principal processando eventos e acionando corrotinas enviando dados a elas. Essa é a ideia básica por trás de `asyncio`, que estudaremos no capítulo 18.

Resumo do capítulo

Segundo Guido van Rossum, há três estilos diferentes de código que você pode escrever usando geradores:

Há o estilo “puxar” (pull) tradicional (iteradores), o estilo “empurrar” (push, como no exemplo de cálculo da média) e há “tarefas” (tasks) (Você já leu o tutorial de corrotinas de Dave Beazley?).¹⁸

18 Mensagem na lista de discussão de “Yield-From: Finalization guarantees” (Yield-From: garantias de finalização, <http://bit.ly/1Jqjn6>) na lista Python-ideas. O tutorial de David Beazley a que Guido se refere é “A Curious Course on Coroutines and Concurrency” (Um curioso curso sobre corrotinas e concorrência, <http://www.dabeaz.com/coroutines/>).

O capítulo 14 foi dedicado aos iteradores; este capítulo apresentou as corrotinas usadas em “estilo empurrar”, e também como “tarefas” bem simples – os processos para táxis no exemplo de simulação. O capítulo 18 as colocará em uso como tarefas assíncronas em programação concorrente.

O exemplo da média cumulativa mostrou um uso comum para uma corrotina: como um acumulador processando itens enviados a ela. Vimos como um decorador pode ser aplicado para preparar (prime) uma corrotina, tornando-a mais conveniente em alguns casos. Contudo tenha em mente que decoradores para preparação (priming decorators) não são compatíveis com alguns usos de corrotinas. Em particular, `yield from subgenerator()` supõe que `subgenerator` não está preparado e prepara-o automaticamente.

As corrotinas acumuladoras podem produzir resultados parciais a cada chamada do método `send`, mas são mais úteis quando podem devolver valores – um recurso adicionado em Python 3.3 com a PEP 380. Vimos como o comando `return the_result` em um gerador agora levanta `StopIteration(the_result)`, permitindo ao chamador recuperar `the_result` do atributo `value` da exceção. É uma maneira bem desajeitada de recuperar resultados de corrotinas, mas é tratado automaticamente pela sintaxe `yield from` introduzida na PEP 380.

A discussão sobre `yield from` começou com exemplos triviais usando iteráveis simples e, em seguida, prosseguiu com um exemplo destacando os três principais componentes de qualquer uso significativo de `yield from`: o gerador delegante (definido pelo uso de `yield from` em seu corpo), o subgerador ativado pelo `yield from` e o código do cliente, que aciona toda essa armação enviando valores ao subgerador pelo canal de passagem estabelecido por `yield from` no gerador delegante. Esta seção encerrou com uma apresentação da definição formal do comportamento de `yield from`, conforme descrito na PEP 380 em língua natural e usando um pseudocódigo semelhante a Python.

Finalizamos o capítulo com o exemplo de simulação de eventos discretos, mostrando como geradores podem ser usados como alternativa às `threads` e `callbacks` para tratar concorrência. Embora simples, a simulação de táxis permitiu uma apreciação inicial de como frameworks orientados a eventos como Tornado e `asyncio` usam um laço principal para acionar corrotinas que executam atividades concorrentes, usando uma única `thread` de execução. Em programação orientada a eventos com corrotinas, cada atividade concorrente é executada por uma corrotina que cede repetidamente o controle de volta ao laço principal, permitindo que outras corrotinas sejam ativadas e prossigam. Essa é uma forma de multitarefa cooperativa: as corrotinas cedem o controle de forma voluntária e explícita ao escalonador central. Em comparação, `threads` implementam multitarefa preemptiva. O escalonador pode suspender uma `thread` a qualquer momento – mesmo no meio de uma instrução – para dar a vez a outra `thread`.

Uma observação final: este capítulo adotou uma definição ampla e informal de corrotina: uma função geradora acionada por um cliente que lhe envia dados por meio de chamadas a `send(...)` ou `yield from`. Essa definição ampla é usada na *PEP 342 – Coroutines via Enhanced Generators* (Corrotinas por meio de geradores melhorados, <https://www.python.org/dev/peps/pep-0342/>) e na maioria dos livros de Python atualmente (quando escrevi este livro). A biblioteca `asyncio` que veremos no capítulo 18 foi desenvolvida com base em corrotinas, porém uma definição mais rigorosa de corrotina foi adotada nessa biblioteca: as corrotinas de `asyncio` são (normalmente) decoradas com `@asyncio.coroutine` e são sempre acionadas por `yield from`, e não pela chamada direta a `.send(...)`. É claro que as corrotinas de `asyncio` são acionadas por `next(...)` e `.send(...)` internamente, mas, em código de usuário, usamos somente `yield from` para executá-las.

Leituras complementares

David Beazley é a autoridade máxima em geradores e corrotinas em Python. O livro *Python Cookbook*, 3E (O'Reilly, <http://shop.oreilly.com/product/0636920027072.do>), escrito em conjunto com Brian Jones, tem várias receitas com corrotinas. Os tutoriais de Beazley em PyCons sobre o assunto são famosos pela profundidade e abrangência. O primeiro foi na PyCon US 2008: “Generator Tricks for Systems Programmers” (Triques com geradores para programadores de sistema, <http://www.dabeaz.com/generators/>). A PyCon US 2009 viu o lendário tutorial “A Curious Course on Coroutines and Concurrency” [Um curioso curso sobre corrotinas e concorrência, <http://www.dabeaz.com/coroutines/>; links de vídeo difíceis de encontrar para as três partes: parte 1 (<http://pyvideo.org/video/213>), parte 2 (<http://pyvideo.org/video/215>) e parte 3 (<http://pyvideo.org/video/214>)]. Seu tutorial mais recente na PyCon 2014 em Montreal foi “Generators: The Final Frontier” (Geradores: a última fronteira, <http://www.dabeaz.com/finalgenerator/>), em que ele apresenta mais exemplos de concorrência – tem realmente mais a ver com os assuntos do capítulo 18 deste livro. Dave não consegue resistir a dar um nó na cabeça das pessoas em suas aulas e, desse modo, na última parte de “The Final Frontier”, as corrotinas substituem o clássico padrão Visitor (Visitante) em um avaliador de expressões aritméticas.

As corrotinas permitem novas maneiras de organizar códigos e, assim como recursão ou polimorfismo (despacho dinâmico), é preciso um pouco de tempo para acostumar-se às suas possibilidades. Um exemplo interessante de um algoritmo clássico reescrito com corrotinas está no post “Greedy algorithm with coroutines” (Algoritmos gulosos com corrotinas, <http://bit.ly/1HG5FQ0>), de James Powell. Você também pode navegar em “Popular recipes tagged *coroutine*” (Receitas populares com tag *coroutine*, <http://bit.ly/1HG5FzA>) no banco de dados de receitas de ActiveState Code (<https://code.activestate.com/recipes/>).

Paul Sokolovsky implementou `yield from` no interpretador extremamente enxuto MicroPython (<http://micropython.org/>) de Damien George, projetado para rodar em microcontroladores. Enquanto estudava o recurso, ele criou um excelente diagrama detalhado (<http://bit.ly/1JlqGxW>) para explicar como `yield from` funciona e compartilhou-o na lista de discussão python-tulip. Sokolovsky foi gentil a ponto de permitir que eu copiasse o PDF para o site deste livro, onde há um URL permanente (<http://flupy.org/resources/yield-from.pdf>).

Quando escrevi este capítulo, a grande maioria dos usos de `yield from` encontrada estava na própria `asyncio` ou em códigos que a utilizavam. Passei bastante tempo procurando exemplos de `yield from` que não dependessem de `asyncio`. Greg Ewing – que escreveu a PEP 380 e implementou `yield from` em CPython – publicou alguns exemplos (<http://bit.ly/1JlqJtu>) de seu uso: uma classe `BinaryTree`, um parser XML simples e um escalonador de tarefas.

O livro *Effective Python* (Addison-Wesley, <http://www.effectivepython.com/>) de Brett Slatkin tem um pequeno capítulo excelente chamado “Consider Coroutines to Run Many Functions Concurrently” (Considere corrotinas para executar muitas funções de forma concorrente, disponibilizado online como capítulo de amostra em <http://bit.ly/1JlqNcZ>). Esse capítulo inclui o melhor exemplo de acionamento de geradores com `yield from` que já vi: uma implementação do Game of Life (<http://bit.ly/1HGskDw>) de John Conway, em que corrotinas são usadas para administrar o estado de cada célula à medida que o jogo avança. O código de exemplo de *Effective Python* pode ser encontrado em um repositório do GitHub (<https://github.com/bslatkin/effectivepython>). Refatorei o código do exemplo de Game of Life – separando as funções e classes que implementam o jogo dos códigos de teste usados no livro de Slatkin (código original em <http://bit.ly/1JlqO0f>). Também reescrevi os testes como doctests, portanto você pode ver a saída de várias corrotinas e classes sem executar o script. O exemplo refatorado (<http://bit.ly/1HGso6j>) está em um gist do GitHub (http://bit.ly/coro_life).

Outros exemplos interessantes de `yield from` sem `asyncio` estão em uma mensagem na lista Python Tutor, “Comparing two CSV files using Python” (Comparando dois arquivos CSV usando Python, <http://bit.ly/1JlqSxf>) de Peter Otten, e em um jogo de Pedra-Papel-Tesoura no tutorial “Iterables, Iterators, and Generators” (Iteráveis, iteradores e geradores, <http://bit.ly/1JlqQ8x>) de Ian Ward, publicado como um notebook de iPython.

Guido van Rossum enviou uma mensagem longa ao grupo do Google python-tulip, cujo título é “The difference between `yield` and `yield from`” (A diferença entre `yield` e `yield from`, <http://bit.ly/1JlqT44>), que vale a pena ler. Nick Coghlan postou uma versão da expansão de `yield from` com muitos comentários em Python-Dev em 21 de março de 2009 (<http://bit.ly/1JlqRcv>); na mesma mensagem, ele escreveu o seguinte:

O fato de pessoas diferentes acharem um código que use `yield from` ser difícil ou não de entender terá mais a ver com o entendimento que elas têm dos conceitos de multitarefa cooperativa em geral do que com os truques subjacentes envolvidos em permitir o uso de geradores realmente aninhados.

A PEP 492 – *Coroutines with `async` and `await` syntax* (Corrotinas com sintaxe `async` e `await`, <https://www.python.org/dev/peps/pep-0492/>) de Yury Selivanov propõe o acréscimo de duas palavras reservadas em Python: `async` e `await`. A primeira será usada com outras palavras reservadas existentes para definir novas construções na linguagem. Por exemplo, `async def` será usada para definir uma corrotina, e `async for` para percorrer iteráveis assíncronos em um laço usando iteradores assíncronos (implementando `__aiter__` e `__anext__`, versões de `__iter__` e `__next__` para corrotinas). Para evitar conflito com a nova palavra reservada `async`, a função essencial `asyncio.async()` será renomeada para `asyncio.ensure_future()` em Python 3.4.4. A palavra reservada `await` fará algo semelhante para `yield from`, mas será permitida somente em corrotinas definidas com `async def` – e nessas corrotinas o uso de `yield` e `yield from` será proibido. Com a nova sintaxe, a PEP estabelece uma clara separação entre os geradores legados que evoluíram para objetos do tipo corrotina e uma nova geração de objetos corrotina nativos, com um suporte melhor da linguagem, graças à infraestrutura como a das palavras reservadas `async` e `await` e a vários métodos especiais novos. As corrotinas estão prestes a se tornar realmente importantes no futuro de Python, e a linguagem deve ser adaptada para melhor integrá-las.

Fazer experimentos com simulações de eventos discretos é uma ótima maneira de se sentir à vontade com multitarefa cooperativa. O artigo “Discrete event simulation” (Simulação de eventos discretos, <http://bit.ly/1JlqXB1>) da Wikipedia é um bom ponto de partida.¹⁹ “Writing a Discrete Event Simulation: Ten Easy Lessons” (Escrevendo uma simulação de eventos discretos: dez lições fáceis, <http://bit.ly/1JlqWgz>) de Ashish Gupta é um pequeno tutorial sobre como escrever simulações de eventos discretos manualmente (sem bibliotecas especiais). O código está em Java, portanto é baseado em classes e não usa corrotinas, mas pode ser facilmente portado para Python. Independentemente do código, o tutorial tem uma boa introdução rápida à terminologia e aos componentes de uma simulação de eventos discretos. Converter os exemplos de Gupta em classes Python e então para classes que aproveitem corrotinas é um bom exercício.

Para uma biblioteca de Python pronta para uso que utilize corrotinas, temos a SimPy. Sua documentação online (<https://simpy.readthedocs.org/en/latest/>) explica o seguinte:

¹⁹ Atualmente, mesmo professores mais experientes concordam que a Wikipedia em inglês é um ótimo lugar para começar a estudar praticamente qualquer assunto em ciência da computação. Não sei sobre outros assuntos, mas, em ciência da computação, a Wikipedia arrasa.

SimPy é um framework para simulações de eventos discretos, baseado em processos, desenvolvido a partir de Python padrão. Seu dispatcher de eventos é baseado em geradores de Python e também pode ser usado para programação assíncrona em rede ou implementação de sistemas com múltiplos agentes (tanto para comunicação simulada quanto real).

Corrotinas não são tão novas em Python, mas estavam muito ligadas a certos nichos de aplicações antes de frameworks de programação assíncrona começarem a tratá-las, começando com o Tornado. O acréscimo de `yield from` em Python 3.3 e de `asyncio` em Python 3.4 provavelmente impulsionará a adoção de corrotinas – e do próprio Python 3.4. No entanto Python 3.4 tem menos de um ano de vida hoje (quando escrevi este capítulo) – portanto, depois de assistir aos tutoriais de David Beazley e ver os exemplos do Cookbook sobre o assunto, não há muitos conteúdos por aí que explorem a programação de corrotinas em Python com muita profundidade. Por enquanto.

Ponto de vista

Raise from lambda

Em linguagens de programação, palavras reservadas estabelecem as regras básicas de controle de fluxo e avaliação de expressões.

Uma palavra reservada em uma linguagem é como uma peça de um jogo de tabuleiro. Na linguagem do xadrez, as palavras reservadas são ♔, ♚, ♝, ♞, ♜ e ♡. No jogo de Go, é ●.

Jogadores de xadrez têm seis tipos de peças diferentes para implementar seus planos, enquanto jogadores de Go aparentemente têm apenas um tipo de peça. Entretanto, na semântica de Go, peças adjacentes formam peças sólidas maiores, de vários formatos diferentes, com propriedades que surgem daí. Algumas formas de organização das peças em Go são indestrutíveis. Go é mais expressivo que xadrez. Em Go, há 361 movimentos possíveis de abertura e um número estimado de $1e+170$ posições permitidas; em xadrez, os números são 20 movimentos de abertura e $1e+50$ posições.

Acrescentar uma nova peça ao xadrez seria uma mudança radical. Adicionar uma nova palavra reservada em uma linguagem de programação também é uma mudança radical. Portanto, para os designers de uma linguagem, faz sentido ser cauteloso quanto à introdução de novas palavras reservadas.

Tabela 16.1 – Número de palavras reservadas em linguagens de programação

Palavras reservadas	Linguagem	Comentário
5	Smalltalk-80	Famosa por sua sintaxe minimalista.
25	Go	A linguagem, e não o jogo.
32	C	Em ANSI C. C99 tem 37 palavras reservadas; C11 tem 44.
33	Python	Python 2.7 tem 31 palavras reservadas; Python 1.5 tinha 28.
41	Ruby	Palavras reservadas podem ser usadas como identificadores (por exemplo, <code>class</code> também é um nome de método).
49	Java	Como em C, os nomes dos tipos primitivos (<code>char</code> , <code>float</code> etc.) são reservados.
60	JavaScript	Inclui todas as palavras reservadas de Java 1.0, muitas das quais não são usadas (http://mzl.la/1J1r8fM).
65	PHP	Desde PHP 5.3 (http://php.net/manual/en/reserved.keywords.php), sete palavras reservadas foram introduzidas, incluindo <code>goto</code> , <code>trait</code> e <code>yield</code> .
85	C++	De acordo com cppreference.com (http://en.cppreference.com/w/cpp/keyword), C++11 acrescentou dez palavras reservadas às 75 existentes.
555	COBOL	Não inventei isso. Veja o manual de ILE COBOL da IBM (http://ibm.co/1J1r7bjf).
∞	Scheme	Qualquer pessoa pode definir novas palavras reservadas.

Python 3 adicionou `nonlocal`, promoveu `None`, `True` e `False` para o status de palavra reservada e removeu `print` e `exec`. É muito incomum que uma linguagem remova palavras reservadas à medida que evolui. A tabela 16.1 lista algumas linguagens, ordenadas pelo número de palavras reservadas.

Scheme herdou de Lisp um recurso de macro que permite a qualquer pessoa criar formas especiais adicionando novas estruturas de controle e regras de avaliação à linguagem. Os identificadores definidos pelo usuário nessas formas são conhecidos como “palavras reservadas sintáticas”. O padrão R5RS de Scheme define que “Não há identificadores reservados” (página 45 do padrão, <http://bit.ly/1J1rB1w>), mas uma implementação típica como MIT/GNU Scheme (<http://bit.ly/1J1rAL1>) vem com 34 palavras reservadas sintáticas predefinidas, como `if`, `lambda` e `define-syntax` – a palavra reservada que permite conjurar novas palavras reservadas.²⁰

Python é como xadrez e Scheme é como Go (o jogo).

Voltemos agora para a sintaxe de Python. Acho que Guido é conservador demais em relação às palavras reservadas. É bom ter um conjunto pequeno delas, e acrescentar novas palavras reservadas pode fazer muitos programas quebrarem.

20 “The Value Of Syntax?” (O valor da sintaxe?, <http://lambda-the-ultimate.org/node/4295>) é uma discussão interessante sobre sintaxe extensível e usabilidade de linguagem de programação. O fórum *Lambda the Ultimate* (<http://lambda-the-ultimate.org/>) é um ponto de encontro para geeks de linguagens de programação.

Porém o uso de `else` em laços revela um problema recorrente: a “sobrecarga” de palavras reservadas existentes quando uma nova palavra seria uma opção melhor. No contexto de `for`, `while` e `try`, uma nova palavra reservada `then` seria preferível a abusar de `else`.

A manifestação mais séria desse problema é a sobrecarga de `def`: ela atualmente é usada para definir funções, geradores e corrotinas – objetos que são diferentes demais para compartilharem a mesma sintaxe de declaração.²¹

A introdução de `yield from` é particularmente preocupante. Mais uma vez, acredito que os usuários de Python estariam mais bem servidos com uma nova palavra reservada. Pior ainda, isso dá início a uma nova tendência: encadear palavras reservadas existentes para criar uma sintaxe nova em vez de acrescentar palavras reservadas descritivas e sensatas. Temo que, um dia, possamos estar quebrando a cabeça para decifrar o significado de `raise from lambda`.

Notícias de última hora

Enquanto concluía o processo de revisão técnica deste livro, parece que a *PEP 492 – Coroutines with async and await syntax* (Corrotinas com sintaxes `async` e `await`, <https://www.python.org/dev/peps/pep-0492/>) de Yury Selivanov estava prestes a ser aceita para implementação já em Python 3.5! A PEP tem o apoio de Guido van Rossum e de Victor Stinner, respectivamente o autor e um mantenedor principal da biblioteca `asyncio`, que seria o caso de uso principal da nova sintaxe. Em resposta à mensagem de Selivanov (<http://bit.ly/1JlrNgY>) em Python-ideas, Guido até mesmo deu pistas de que o lançamento de Python 3.5 pode atrasar (<http://bit.ly/1JlrPp9>) para que a PEP possa ser implementada.

É claro que, com isso, a maior parte das reclamações que expressei nas seções anteriores poderia ser deixada de lado.

²¹ Um post altamente recomendado relacionado a essa questão no contexto de JavaScript, Python e outras linguagens é “What Color Is Your Function?” (Qual é a cor de sua função?, <http://bit.ly/1Jlrldh>) de Bob Nystrom.

CAPÍTULO 17

Concorrência com futures

Pessoas que criticam threads normalmente são programadores de sistemas que têm em mente casos de uso que um programador de aplicações típico jamais encontrará em sua vida. [...] Em 99% dos casos de uso que um programador de aplicações pode encontrar, o simples padrão de disparar um punhado de threads independentes e coletar os resultados em uma fila é tudo que uma pessoa precisa saber.¹

— Michele Simionato

Pensador da linguagem Python

O foco deste capítulo é a biblioteca `concurrent.futures`, introduzida em Python 3.2, mas disponível também para Python 2.5 e versões mais recentes como o pacote `futures` (<https://pypi.python.org/pypi/futures/>) no PyPI. Essa biblioteca encapsula o padrão descrito por Michele Simionato na citação anterior, tornando seu uso quase trivial.

Introduzirei também o conceito de “futures” (futuros) – objetos que representam a execução assíncrona de uma operação. Essa ideia poderosa é a base não só de `concurrent.futures`, mas também do pacote `asyncio`, que discutiremos no capítulo 18.

Começaremos com um exemplo para motivação.

Exemplo: downloads da Web em três estilos

Para tratar E/S em rede de modo eficiente, você precisa de concorrência, pois a rede introduz alta latência – portanto, em vez de desperdiçar ciclos de CPU esperando, é melhor fazer outra coisa até que uma resposta seja recebida da rede.

¹ Do post *Threads, processes and concurrency in Python: some thoughts* (Threads, processos e concorrência em Python: algumas ideias, <http://bit.ly/1JlrYZQ>) de Michele Simionato, cujo subtítulo é “Removing the hype around the multicore (non) revolution and some (hopefully) sensible comment about threads and other forms of concurrency” (Acabando com o rebuliço em torno da (não) revolução de múltiplos cores e alguns comentários (espero que) sensatos sobre threads e outras formas de concorrência).

Para ilustrar essa última questão por meio de código, escrevi três programas simples para fazer download de vinte bandeiras de países publicadas em um website. O primeiro programa, *flags.py*, executa sequencialmente: só requisita a próxima imagem quando a imagem anterior já estiver baixada e salva em disco. Os outros dois scripts fazem downloads concorrentes: requisitam todas as imagens praticamente ao mesmo tempo e salvam os arquivos à medida que vão chegando. O script *flags_threadpool.py* usa o pacote `concurrent.futures`, enquanto *flags_asyncio.py* usa `asyncio`.

O exemplo 17.1 mostra o resultado da execução dos três scripts, três vezes cada. Também postei um vídeo de 73 segundos no YouTube (<https://www.youtube.com/watch?v=A9e9Cy1UkME>) para que você possa vê-los executando enquanto uma janela do Finder no OS X exibe as bandeiras à medida que são salvas. Os scripts fazem download de imagens de *flupy.org*, que está por trás de uma CDN, portanto você poderá ver resultados mais lentos nas primeiras execuções. Os resultados no exemplo 17.1 foram obtidos após várias execuções, portanto o cache da CDN estava aquecido.

Exemplo 17.1 – Três execuções típicas dos scripts *flags.py*, *flags_threadpool.py* e *flags_asyncio.py*

```
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN ❶
20 flags downloaded in 7.26s ❷
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.20s
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.09s
$ python3 flags_threadpool.py
DE BD CN JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
20 flags downloaded in 1.37s ❸
$ python3 flags_threadpool.py
EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CN VN ET IR
20 flags downloaded in 1.60s
$ python3 flags_threadpool.py
BD DE EG CN ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
20 flags downloaded in 1.22s
$ python3 flags_asyncio.py ❹
BD BR IN ID TR DE CN US IR PK PH FR RU NG VN ET MX EG JP CD
20 flags downloaded in 1.36s
```

```
$ python3 flags_asyncio.py
RU CN BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
20 flags downloaded in 1.27s
$ python3 flags_asyncio.py
RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CN JP PH CD TR ❸
20 flags downloaded in 1.42s
```

- ❶ A saída de cada execução começa com os códigos dos países cujas bandeiras estão sendo baixadas e termina com uma mensagem informando o tempo decorrido.
- ❷ *flags.py* demorou uma média de 7,18s para fazer download de vinte imagens.
- ❸ A média de *flags_threadpool.py* foi de 1,40s.
- ❹ Para *flags_asyncio.py*, 1,35s foi o tempo médio.
- ❺ Observe a ordem dos códigos dos países: com os scripts concorrentes, os downloads ocorreram em ordem diferente a cada vez.

A diferença de desempenho entre os scripts concorrentes não é significativa, mas ambos são mais de cinco vezes mais rápidos que o script sequencial – e isso apenas para uma tarefa razoavelmente pequena. Se você escalar a tarefa para centenas de downloads, os scripts concorrentes poderão ser mais rápidos que o script sequencial por um fator de vinte ou mais.



Quando testar clientes HTTP concorrentes na Web pública, você poderá inadvertidamente iniciar um ataque DoS (denial-of-service, ou negação de serviço) ou ser considerado suspeito de fazer isso. No caso do exemplo 17.1, não há problemas, pois o código desses scripts está fixo para fazer apenas vinte requisições. Para testar clientes HTTP não triviais, configure seu próprio servidor de testes. O arquivo *17-futures/countries/README.rst* (<http://bit.ly/1Jlsg2L>) no repositório de código de *Python fluente* no GitHub (<https://github.com/fluentpython/example-code>) tem instruções para configurar um servidor Nginx local.

Vamos agora estudar as implementações de dois dos scripts testados no exemplo 17.2: *flags.py* e *flags_threadpool.py*. Deixarei o terceiro script, *flags_asyncio.py*, para o capítulo 18, mas demonstrei os três juntos para enfatizar uma questão: independentemente da estratégia de concorrência usada – threads ou asyncio –, você verá uma taxa de transferência de dados (throughput) muito melhor nesses códigos em comparação com um código sequencial em aplicações limitadas por E/S (I/O bound), se você codá-las apropriadamente.

Vamos ver o código.

Um script para download sequencial

O exemplo 17.2 não é muito interessante, mas reutilizaremos a maior parte de seu código e as configurações para implementar os scripts concorrentes, portanto ele merece um pouco de atenção.



Por questões de clareza, não há tratamento de erros no exemplo 17.2. Lidaremos com as exceções depois, mas aqui queremos dar enfoque na estrutura de código básica para facilitar a comparação desse script com os scripts concorrentes.

Exemplo 17.2 — flags.py: script para download sequencial; algumas funções serão reutilizadas pelos outros scripts

```
import os
import time
import sys

import requests ❶
POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split() ❷

BASE_URL = 'http://flupy.org/data/flags' ❸
DEST_DIR = 'downloads/' ❹

def save_flag(img, filename): ❺
    path = os.path.join(DEST_DIR, filename)
    with open(path, 'wb') as fp:
        fp.write(img)

def get_flag(cc): ❻
    url = '{}/{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = requests.get(url)
    return resp.content

def show(text): ❼
    print(text, end=' ')
    sys.stdout.flush()

def download_many(cc_list): ❽
    for cc in sorted(cc_list): ❾
        image = get_flag(cc)
        show(cc)
        save_flag(image, cc.lower() + '.gif')
    return len(cc_list)
```

```
def main(download_many): ❩
    t0 = time.time()
    count = download_many(POP20_CC)
    elapsed = time.time() - t0
    msg = '\n{} flags downloaded in {:.2f}s'
    print(msg.format(count, elapsed))

if __name__ == '__main__':
    main(download_many) ❫
```

- ❶ Importa a biblioteca `requests`; ela não faz parte da biblioteca-padrão, portanto, por convenção, nós a importamos após os módulos `os`, `time` e `sys` da biblioteca-padrão e a separamos com uma linha em branco.
- ❷ Lista dos códigos ISO 3166 para os vinte países mais populosos em ordem decrescente de população.
- ❸ O site com as imagens das bandeiras.²
- ❹ Diretório local em que as imagens são salvas.
- ❺ Simplesmente salva `img` (uma sequência de bytes) em `filename` no diretório `DEST_DIR`.
- ❻ Dado um código de país, cria o URL e faz download da imagem, devolvendo o conteúdo binário da resposta.
- ❼ Exibe uma string e descarrega (faz um flush) `sys.stdout` para que possamos ver o progresso em uma linha; isso é necessário porque Python normalmente espera uma quebra de linha para descarregar o buffer `stdout`.
- ❽ `download_many` é a função principal a ser comparada com as implementações concorrentes.
- ❾ Percorre a lista de códigos de países em ordem alfabética para deixar claro que a ordem é preservada na saída; devolve a quantidade de códigos de países cujas bandeiras foram baixadas.
- ❩ `main` registra e informa o tempo decorrido após executar `download_many`.
- ❫ `main` deve ser chamado com a função que fará os downloads; passamos a função `download_many` como argumento para que `main` possa ser usado como uma função de biblioteca com outras implementações de `download_many` nos próximos exemplos.

2 As imagens são originalmente de *CIA World Factbook* (<http://1.usa.gov/1JsmHJ>), uma publicação de domínio público do governo norte-americano. Copiei-as para o meu site para evitar o risco de iniciar um ataque DoS em `CIA.gov`.



A biblioteca `requests` de Kenneth Reitz está disponível em PyPI (<https://pypi.python.org/pypi/requests>) e é mais poderosa e mais fácil de usar que o módulo `urllib.request` da biblioteca-padrão de Python 3. De fato, `requests` é considerada um modelo de API pythônica. Ela também é compatível com Python 2.6 e versões mais recentes, enquanto `urllib2` de Python 2 foi movida e renomeada em Python 3, portanto é mais conveniente usar `requests`, independentemente da versão de Python que você pretende usar.

Não há nada realmente novo em `flags.py`. Serve como uma base de comparação para os demais scripts, e usei-o como uma biblioteca para evitar código redundante ao implementá-los. Vamos agora ver uma reimplementação usando `concurrent.futures`.

Fazendo download com `concurrent.futures`

Os principais recursos do pacote `concurrent.futures` são as classes `ThreadPoolExecutor` e `ProcessPoolExecutor`; elas implementam uma interface que permite que você submeta invocáveis para serem executados em threads ou em processos diferentes, respectivamente. As classes administram um pool interno de threads ou processos de trabalho, além de uma fila de tarefas a serem executadas. Mas é uma interface de alto nível, e não precisamos conhecer nenhum desses detalhes para um caso de uso simples como nossos downloads de bandeiras.

O exemplo 17.3 mostra a maneira mais simples de implementar os downloads de modo concorrente usando o método `ThreadPoolExecutor.map`.

Exemplo 17.3 – `flags_threadpool.py`: script de download com threads usando `futures.ThreadPoolExecutor`

```
from concurrent import futures

from flags import save_flag, get_flag, show, main ❶

MAX_WORKERS = 20 ❷

def download_one(cc): ❸
    image = get_flag(cc)
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc

def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list)) ❹
    with futures.ThreadPoolExecutor(workers) as executor: ❺
        res = executor.map(download_one, sorted(cc_list)) ❻
```



```
return len(list(res))❷
```

```
if __name__ == '__main__':  
    main(download_many) ❸
```

- ❶ Reutiliza algumas funções do módulo `flags` (Exemplo 17.2).
- ❷ Número máximo de threads a serem usadas em `ThreadPoolExecutor`.
- ❸ Função para download de uma única imagem: é o que cada thread executará.
- ❹ Define o número de threads de trabalho: usa o menor número entre o máximo que queremos permitir (`MAX_WORKERS`) e os itens a serem processados; desse modo, nenhuma thread desnecessária será criada.
- ❺ Instancia `ThreadPoolExecutor` com essa quantidade de threads de trabalho; o método `executor.__exit__` chamará `executor.shutdown(wait=True)`, que ficará bloqueado até que todas as threads terminem.
- ❻ O método `map` é semelhante à função embutida `map`, exceto que a função `download_one` será chamada de forma concorrente a partir de várias threads; ele devolve um gerador em que você pode iterar para recuperar o valor devolvido pelas funções.
- ❼ Devolve o número de resultados obtidos: se alguma das chamadas com thread levantar uma exceção, essa exceção será levantada aqui, quando a chamada implícita a `next()` do construtor da lista tentar recuperar o valor de retorno correspondente do iterador.
- ❽ Chama a função `main` do módulo `flags`, passando a versão melhorada de `download_many`.

Observe que a função `download_one` do exemplo 17.3 é essencialmente o corpo do laço `for` da função `download_many` do exemplo 17.2. Essa é uma refatoração comum quando escrevemos código concorrente: transformar o corpo de um laço `for` sequencial em uma função a ser chamada de forma concorrente.

A biblioteca se chama `concurrency.futures`, apesar de não vermos nenhum objeto `future` no exemplo 17.3, portanto você deve estar se perguntando onde estão eles. A próxima seção explica isso.

Onde estão os futures?

Futures são componentes essenciais no funcionamento interno de `concurrent.futures` e de `asyncio`, mas, como usuários dessas bibliotecas, às vezes, não os vemos. O exemplo 17.3 utiliza futures internamente, mas o código que escrevi não entra em contato direto com eles. Esta seção apresenta uma visão geral dos futures, com um exemplo mostrando-os em ação.

Em Python 3.4, há duas classes chamadas `Future` na biblioteca-padrão: `concurrent.futures.Future` e `asyncio.Future`. Elas têm a mesma finalidade: uma instância de qualquer uma das classes `Future` representa um processamento adiado que pode ou não ter sido concluído. É semelhante à classe `Deferred` em Twisted, à classe `Future` em Tornado e aos objetos `Promise` em várias bibliotecas de JavaScript.

Futures encapsulam operações pendentes para que elas possam ser colocadas em filas, seus estados de completude possam ser consultados e seus resultados (ou exceções) possam ser recuperados quando estiverem disponíveis.

Um aspecto importante a saber sobre futures em geral é que você e eu não devemos criá-los: eles foram concebidos para serem instanciados exclusivamente pelo framework de concorrência, seja `concurrent.futures` ou `asyncio`. É fácil entender por quê: um `Future` representa algo que acontecerá no futuro, e a única maneira de ter certeza de que algo acontecerá é agendando sua execução. Desse modo, instâncias de `concurrent.futures.Future` são criadas apenas como resultado de agendar algo para execução com uma subclasse de `concurrent.futures.Executor`. Por exemplo, o método `Executor.submit()` aceita um invocável, agenda-o para ser executado e devolve um `Future`.

O código do cliente não deve alterar o estado de um `Future`: o framework de concorrência muda o estado de um `Future` quando o processamento que ele representa termina, e não podemos controlar quando isso acontece.

Ambos os tipos de `Future` têm um método `.done()` que é não bloqueante e devolve um booleano informando se o invocável associado a esse `Future` executou ou não. Em vez de perguntar se um `Future` foi executado, o código do cliente normalmente pede para ser notificado. É por isso que ambas as classes `Future` têm um método `.add_done_callback()`: você lhe fornece um invocável e ele será chamado com o `Future` como único argumento quando esse `Future` tiver sido executado.

Há também um método `.result()` que funciona do mesmo jeito para ambas as classes quando o `Future` é executado: devolve o resultado do invocável ou levanta novamente qualquer exceção que possa ter sido lançada quando o invocável foi executado. No entanto, quando o `Future` ainda não executou, o comportamento do método `result` é bem diferente entre as duas variantes de `Future`. Em uma instância de `concurrency.futures.Future`, chamar `f.result()` bloqueará a thread que fez a chamada até o resultado estar pronto. Um argumento `timeout` opcional pode ser passado e, se o `Future` não for executado no tempo especificado, uma exceção `TimeoutError` será levantada. Na seção “`asyncio.Future`: não bloqueante por design” na página 603, veremos que o método `asyncio.Future.result` não aceita `timeout`, e a maneira preferível de obter o resultado de futures naquela biblioteca é usar `yield from` – que não funciona com instâncias de `concurrency.futures.Future`.

Várias funções em ambas bibliotecas devolvem futures; outras os usam em suas implementações de forma transparente ao usuário. Um exemplo do último caso é `Executor.map`, que vimos no exemplo 173: ele devolve um iterador em que `__next__` chama o método `result` de cada future, assim o que obtemos são os resultados dos futures, e não os futures em si.

Para uma visão prática dos futures, podemos reescrever o exemplo 173 usando a função `concurrent.futures.as_completed`, que aceita um iterável de futures e devolve um iterador que produz futures à medida que terminam.

Usar `futures.as_completed` exige mudanças somente na função `download_many`. A chamada de alto nível a `executor.map` é substituída por dois laços `for`: um para criar e agendar futures e outro para obter seus resultados. Aproveitando a ocasião, acrescentaremos algumas chamadas a `print` para exibir cada future antes e depois de sua execução. O exemplo 174 mostra o código para uma nova função `download_many`. O código de `download_many` cresceu de 5 para 17 linhas, mas agora podemos inspecionar os misteriosos futures. As funções restantes são as mesmas do exemplo 173.

Exemplo 17.4 – `flags_threadpool_ac.py`: substituindo `executor.map` por `executor.submit` e `futures.as_completed` na função `download_many`

```
def download_many(cc_list):
    cc_list = cc_list[:5] ❶
    with futures.ThreadPoolExecutor(max_workers=3) as executor: ❷
        to_do = []
        for cc in sorted(cc_list): ❸
            future = executor.submit(download_one, cc) ❹
            to_do.append(future) ❺
            msg = 'Scheduled for {}: {}'.format(cc, future)
            print(msg) ❻

    results = []
    for future in futures.as_completed(to_do): ❼
        res = future.result() ❽
        msg = '{} result: {}'.format(future, res)
        print(msg) ❾
        results.append(res)

    return len(results)
```

- ❶ Nessa demonstração, usamos apenas os cinco países mais populosos.
- ❷ Fixa `max_workers` com 3 para que possamos observar futures pendentes na saída.
- ❸ Faz uma iteração pelos códigos dos países em ordem alfabética para deixar claro que os resultados chegam fora de ordem.

- ❶ `executor.submit` agenda o invocável a ser executado e devolve um `future` que representa essa operação pendente.
- ❷ Armazena cada `future` para que possamos recuperá-lo depois com `as_completed`.
- ❸ Exibe uma mensagem com o código do país e o respectivo `future`.
- ❹ `as_completed` produz futures à medida que são concluídos.
- ❺ Obtém o resultado desse `future`.
- ❻ Exibe o `future` e seu resultado.

Observe que a chamada a `future.result()` nunca vai bloquear nesse exemplo, pois o `future` é produzido por `as_completed`. O exemplo 17.5 mostra a saída de uma execução do exemplo 17.4.

Exemplo 17.5 – Saída de `flags_threadpool_ac.py`

```
$ python3 flags_threadpool_ac.py
Scheduled for BR: <Future at 0x100791518 state=running> ❶
Scheduled for CN: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending> ❷
Scheduled for US: <Future at 0x101807128 state=pending>
CN <Future at 0x100791710 state=finished returned str> result: 'CN' ❸
BR ID <Future at 0x100791518 state=finished returned str> result: 'BR' ❹
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result: 'IN'
US <Future at 0x101807128 state=finished returned str> result: 'US'

5 flags downloaded in 0.70s
```

- ❶ Os futures são agendados em ordem alfabética; a `repr()` de um `future` mostra seu estado: os três primeiros são `running` porque há três `threads` de trabalho.
- ❷ Os dois últimos futures estão no estado `pending`, à espera de `threads` de trabalho.
- ❸ O primeiro CN aqui é a saída de `download_one` em uma `thread` de trabalho; o restante da linha é a saída de `download_many`.
- ❹ Nesse ponto, duas `threads` mostram códigos antes que `download_many` na `thread` principal possa exibir o resultado da primeira `thread`.



Se executar `flags_threadpool_ac.py` várias vezes, você verá a ordem dos resultados variar. Aumentar o valor do argumento `max_workers` para 5 aumentará a variação na ordem dos resultados. Reduzir seu valor para 1 fará esse código executar sequencialmente e a ordem dos resultados sempre será a ordem das chamadas a `submit`.

Vimos duas variantes do script de download usando `concurrent.futures`: o exemplo 17.3 com `ThreadPoolExecutor.map` e o exemplo 17.4 com `futures.as_completed`. Se estiver curioso sobre o código de *flags_asyncio.py*, dê uma olhada no exemplo 18.5 do capítulo 18.

Estritamente falando, nenhum dos scripts concorrentes que testamos até agora é capaz de fazer downloads em paralelo. Estes exemplos com `concurrent.futures` são limitados pela GIL, e *flags_asyncio.py* tem uma só thread (é single-threaded).

A essa altura, talvez você tenha perguntas sobre os benchmarks informais que acabamos de fazer:

- Como *flags_threadpool.py* é capaz de executar cinco vezes mais rápido que *flags.py* se as threads em Python estão limitadas por uma GIL (Global Interpreter Lock, ou Trava Global do Interpretador), que permite apenas uma thread executando a qualquer momento?
- Como *flags_asyncio.py* é capaz de executar cinco vezes mais rápido que *flags.py* quando ambos têm uma só thread?

Responderei à segunda pergunta na seção “Dando voltas em chamadas bloqueantes” na página 610.

Continue lendo para entender por que a GIL é quase inofensiva em processamento limitado por E/S (I/O bound).

E/S bloqueante e a GIL

As estruturas de dados internas do interpretador CPython não são thread-safe (seguras para uso com threads), por isso existe uma GIL (Global Interpreter Lock, ou Trava Global do Interpretador), que permite apenas uma thread de cada vez executar bytecodes Python. É por isso que um único processo Python normalmente não pode usar vários núcleos de CPU ao mesmo tempo.³

Quando escrevemos código Python, não temos controle sobre a GIL, mas uma função embutida ou uma extensão implementada em C pode liberá-la quando executa uma tarefa que consome muito tempo. De fato, uma biblioteca Python escrita em C pode administrar a GIL, iniciar suas próprias threads de sistema operacional e aproveitar todos os cores de CPU disponíveis. Isso complica consideravelmente o código da biblioteca, e a maioria dos autores de bibliotecas não faz isso.

No entanto todas as funções da biblioteca-padrão que realizam E/S bloqueante liberam a GIL enquanto esperam um resultado do sistema operacional. Isso quer dizer

³ Essa é uma limitação do interpretador CPython, e não da linguagem Python em si. Jython e IronPython não têm essa limitação, mas Pypy, que é o interpretador Python mais rápido disponível, também tem uma GIL.

que programas Python limitados por E/S podem se beneficiar do uso de threads no nível de Python: enquanto uma thread Python espera uma resposta da rede, a função de E/S bloqueada libera a GIL para outra thread executar.

É por isso que David Beazley diz: “Threads em Python são ótimas para não fazer nada.”⁴



Toda função bloqueante de E/S da biblioteca-padrão de Python libera a GIL, permitindo que outras threads executem. A função `time.sleep()` também libera a GIL. Desse modo, as threads de Python são perfeitamente utilizáveis em aplicações limitadas por E/S, apesar da GIL.

Vamos agora ver rapidamente uma maneira simples de contornar a GIL para tarefas limitadas por CPU (CPU-bound) usando `concurrent.futures`.

Iniciando processos com `concurrent.futures`

A página de documentação de `concurrent.futures` (<https://docs.python.org/3/library/concurrent.futures.html>) tem como subtítulo “Launching parallel tasks” (Iniciando tarefas paralelas). O pacote realmente permite o processamento paralelo, pois tem suporte para distribuição de tarefas entre vários processos Python usando a classe `ProcessPoolExecutor` – contornando assim a GIL e aproveitando todos os cores de CPU disponíveis, se você precisar executar processamento intensivo em CPU.

Tanto `ProcessPoolExecutor` quanto `ThreadPoolExecutor` implementam a interface genérica de `Executor`, portanto é muito fácil mudar de uma solução baseada em threads para uma baseada em processos usando `concurrent.futures`.

Não há vantagens em usar `ProcessPoolExecutor` para o exemplo de download de bandeiras ou qualquer tarefa limitada por E/S. É fácil comprovar isso; basta mudar as seguintes linhas do exemplo 173:

```
def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))
    with futures.ThreadPoolExecutor(workers) as executor:
```

para:

```
def download_many(cc_list):
    with futures.ProcessPoolExecutor() as executor:
```

Para usos simples, a única diferença perceptível entre as duas classes executoras concretas é que `ThreadPoolExecutor.__init__` exige um argumento `max_workers` para definir o número de threads no pool. É um argumento opcional em `ProcessPoolExecutor` e,

⁴ Slide 106 de “Generators: The Final Frontier” (Geradores: a última fronteira, <http://www.dabeaz.com/finalgenerator/>).

na maioria das vezes, não o usamos – o default é o número de CPUs devolvido por `os.cpu_count()`. Isso faz sentido: para processamento limitado por CPU, é inútil solicitar mais processos de trabalho que o número de CPUs. Por outro lado, para processamento limitado por E/S, podemos usar 10, 100 ou 1.000 threads em um `ThreadPoolExecutor`; o melhor número dependerá do que você estiver fazendo e da memória disponível, e encontrar o número ideal exigirá testes cuidadosos.

Alguns testes mostraram que o tempo médio de download das vinte bandeiras aumentou para 1,8s com um `ProcessPoolExecutor` – em comparação com os 1,4s da versão original com `ThreadPoolExecutor`. O principal motivo provavelmente é o limite de quatro downloads concorrentes em meu computador com quatro cores, contra vinte threads de trabalho na versão com pool de threads.

A vantagem de `ProcessPoolExecutor` está em tarefas que fazem uso intensivo de CPU. Realizei alguns testes de desempenho com dois scripts limitados por CPU:

arcfour_futures.py

Criptografa e descriptografa uma dúzia de arrays de bytes com tamanhos de 149 KB a 384 KB usando uma implementação do algoritmo RC4 em Python puro (listagem no exemplo A.7).

sha_futures.py

Calcula o hash SHA-256 de uma dúzia de arrays de bytes de 1 MB com o pacote `hashlib` da biblioteca-padrão, que usa a biblioteca OpenSSL (listagem no exemplo A.9).

Nenhum desses scripts executa E/S, exceto para exibir os resultados agregados. Eles criam e processam todos os seus dados em memória, portanto E/S não interfere em seus tempos de execução.

A tabela 17.1 mostra os tempos médios que obtive após 64 execuções do exemplo com RC4 e 48 execuções do exemplo com SHA. As medições incluem o tempo gasto para criação (spawn) dos processos de trabalho.

Tabela 17.1 – Tempo e fator de ganho de velocidade para os exemplos com RC4 e SHA, com um a quatro processos de trabalho em um computador de quatro cores Intel Core i7 de 2.7 GHz usando Python 3.4

Processos de trabalho	Tempo para RC4	Fator para RC4	Tempo para SHA	Fator para SHA
1	11,48s	1,00x	22,66s	1,00x
2	8,65s	1,33x	14,90s	1,52x
3	6,04s	1,90x	11,91s	1,90x
4	5,58s	2,06x	10,89s	2,08x

Em suma, para algoritmos de criptografia, você pode esperar o dobro do desempenho ao iniciar quatro processos de trabalho com um `ProcessPoolExecutor`, se tiver quatro cores de CPU.

Para o exemplo em Python puro de RC4, você pode obter resultados 3,8 vezes mais rápidos se usar PyPy e quatro processos de trabalho, quando comparado com CPython e quatro processos de trabalho. É um aumento de velocidade de 7,8 vezes em relação à base de referência, com um processo de trabalho e CPython na tabela 17.1.



Se você estiver executando uma tarefa com uso intensivo de CPU em Python, experimente usar PyPy (<http://pypy.org/>). O exemplo `arcfour_futures.py` executou de 3,8 a 5,1 vezes mais rápido com PyPy, dependendo do número de processos de trabalho usados. Testei com PyPy 2.4.0, que é compatível com Python 3.2.5, portanto tem `concurrent.futures` na biblioteca-padrão.

Vamos agora investigar o comportamento de um pool de threads com um programa de demonstração que inicia um pool com três threads de trabalho, executando cinco invocáveis que apresentam mensagens com timestamp.

Fazendo experimentos com `Executor.map`

A maneira mais simples de executar vários invocáveis de forma concorrente é com a função `Executor.map`, que vimos pela primeira vez no exemplo 17.3. O exemplo 17.6 tem um script que mostra o funcionamento de `Executor.map` com alguns detalhes. Sua saída está no exemplo 17.7.

Exemplo 17.6 – `demo_executor_map.py`: demonstração simples do método `map` de `ThreadPoolExecutor`

```
from time import sleep, strftime
from concurrent import futures

def display(*args): ❶
    print(strftime('%H:%M:%S'), end=' ')
    print(*args)

def loiter(n): ❷
    msg = '{}loiter({}): doing nothing for {}s...'
    display(msg.format('\t'*n, n, n))
    sleep(n)
    msg = '{}loiter({}): done.'
    display(msg.format('\t'*n, n))
    return n * 10 ❸
```

```
def main():
    display('Script starting.')
    executor = futures.ThreadPoolExecutor(max_workers=3) ❹
    results = executor.map(loiter, range(5)) ❺
    display('results:', results) # ❻.
    display('Waiting for individual results:')
    for i, result in enumerate(results): ❼
        display('result {}: {}'.format(i, result))

main()
```

- ❶ Esta função simplesmente exibe os argumentos que receber, precedidos por um timestamp no formato [HH:MM:SS].
- ❷ loiter não faz nada, exceto exibir uma mensagem quando inicia, dorme por n segundos e então exibe uma mensagem quando termina; tabulações são usadas para indentar as mensagens de acordo com o valor de n .
- ❸ loiter devolve $n * 10$ para que possamos ver como coletar resultados.
- ❹ Cria um ThreadPoolExecutor com três threads.
- ❺ Submete cinco tarefas ao executor (por haver apenas três threads, somente três dessas tarefas iniciarão imediatamente: as chamadas a loiter(0), loiter(1) e loiter(2)); essa é uma chamada não bloqueante.
- ❻ Exibe o resultado da chamada a executor.map imediatamente: é um gerador, como mostra a saída no exemplo 17.7.
- ❼ A chamada a enumerate no laço for chama implicitamente next(results), que, por sua vez, chamará _f.result() no future _f (interno) que representa a primeira chamada, loiter(0). O método result ficará bloqueado até que o future tenha terminado; desse modo, cada iteração nesse laço precisará esperar até o próximo resultado estar pronto.

Incentivo você a executar o exemplo 17.6 e ver a tela ser atualizada de forma incremental. Quando fizer isso, brinque com o argumento `max_workers` de `ThreadPoolExecutor` e com a função `range` que gera os argumentos da chamada a `executor.map` – ou substitua por listas de valores selecionados manualmente para criar diferentes atrasos.

O exemplo 17.7 apresenta uma execução do exemplo 17.6.

Exemplo 17.7 – Amostra da execução de `demo_executor_map.py` do exemplo 17.6

```
$ python3 demo_executor_map.py
[15:56:50] Script starting. ❶
[15:56:50] loiter(0): doing nothing for 0s... ❷
```

```

[15:56:50] loiter(0): done.
[15:56:50]      loiter(1): doing nothing for 1s... ❶
[15:56:50]      loiter(2): doing nothing for 2s...
[15:56:50] results: <generator object result_iterator at 0x106517168> ❷
[15:56:50]      loiter(3): doing nothing for 3s... ❸
[15:56:50] Waiting for individual results:
[15:56:50] result 0: 0 ❹
[15:56:51]      loiter(1): done. ❺
[15:56:51]      loiter(4): doing nothing for 4s...
[15:56:51] result 1: 10 ❻
[15:56:52]      loiter(2): done. ❼
[15:56:52] result 2: 20
[15:56:53]      loiter(3): done.
[15:56:53] result 3: 30
[15:56:55]      loiter(4): done. ❽
[15:56:55] result 4: 40

```

- ❶ Essa execução iniciou em 15:56:50.
- ❷ A primeira thread executa `loiter(0)`, portanto dormirá por 0s e retornará mesmo antes de a segunda thread ter a chance de iniciar, mas você pode ter uma experiência diferente.⁵
- ❸ `loiter(1)` e `loiter(2)` começam imediatamente (como o pool de threads tem três threads de trabalho, três funções podem ser executadas de forma concorrente).
- ❹ Isso mostra que `results` devolvido por `executor.map` é um gerador; nada até agora ficará bloqueado, independentemente do número de tarefas e da configuração de `max_workers`.
- ❺ Como `loiter(0)` terminou, a primeira thread de trabalho agora está disponível para iniciar a quarta thread para `loiter(3)`.
- ❻ É aqui que a execução pode ficar bloqueada, dependendo dos parâmetros fornecidos às chamadas a `loiter`: o método `__next__` do gerador `results` precisa esperar até que o primeiro future termine. Nesse caso, ele não ficará bloqueado porque a chamada a `loiter(0)` terminou antes de esse laço começar. Observe que tudo até esse ponto ocorreu no mesmo segundo do relógio: 15:56:50.
- ❼ `loiter(1)` terminou um segundo depois, em 15:56:51. A thread está livre para iniciar `loiter(4)`.

5 Com threads, não sabemos a sequência exata de eventos que devem ocorrer praticamente no mesmo instante; é possível que, em outro computador, você veja `loiter(1)` começar antes de `loiter(0)` terminar, particularmente porque `sleep` sempre libera a GIL, portanto Python pode alternar para outra thread, mesmo que você durma por 0 segundo.

- ⑧ O resultado de `loiter(1)` é mostrado: 10. Agora o laço `for` ficará bloqueado à espera do resultado de `loiter(2)`.
- ⑨ O padrão se repete: `loiter(2)` termina, seu resultado é exibido; o mesmo vale para `loiter(3)`.
- ⑩ Há um atraso de 2s até `loiter(4)` terminar porque ele começou em 15:56:51 e não fez nada por 4s.

A função `Executor.map` é fácil de usar, mas tem um recurso que pode ou não ser útil, conforme suas necessidades: ela devolve os resultados exatamente na mesma ordem em que as chamadas foram iniciadas; se a primeira chamada demorar 10s para produzir um resultado, e as demais demorarem 1s cada, seu código ficará bloqueado por 10s, pois tentará obter o primeiro resultado do gerador devolvido por `map`. Depois disso, você terá os resultados restantes sem que haja bloqueio, pois já estarão prontos. Não haverá problemas se você precisar de todos os resultados antes de prosseguir, mas, muitas vezes, é preferível obter os resultados à medida que ficarem prontos, independentemente da ordem em que foram submetidos. Para isso, você precisa de uma combinação do método `Executor.submit` com a função `futures.as_completed`, como vimos no exemplo 174. Voltaremos a essa técnica na seção “Usando `futures.as_completed`” na página 584.



A combinação de `executor.submit` com `futures.as_completed` é mais flexível que `executor.map` porque você pode submeter (com `submit`) diferentes invocáveis e argumentos, enquanto `executor.map` foi projetado para executar o mesmo invocável com argumentos diferentes. Além disso, o conjunto de `futures` que você passa para `futures.as_completed` pode ser proveniente de mais de um executor – pode ser que alguns tenham sido criados por uma instância de `ThreadPoolExecutor`, enquanto outros são de um `ProcessPoolExecutor`.

Na próxima seção, retomaremos os exemplos de download de bandeiras com novos requisitos que nos forçarão a iterar pelos resultados de `futures.as_completed` em vez de usar `executor.map`.

Downloads com exibição de progresso e tratamento de erros

Como mencionamos, os scripts da seção “Exemplo: downloads da Web em três estilos” na página 560 não têm tratamento de erro para que seja mais fácil ler e comparar a estrutura das três abordagens: sequencial, com `threads` e assíncrona.

Para testar o tratamento de várias condições de erro, criei os exemplos da série `flags2`:

flags2_common.py

Esse módulo contém funções e configurações comuns usadas por todos os exemplos da série *flags2*, incluindo uma função *main*, que cuida da análise da linha de comando, das medições de tempo e da apresentação dos resultados. É realmente um código auxiliar, não diretamente relevante ao assunto deste capítulo, portanto o código-fonte está no Apêndice A, no exemplo A.10.

flags2_sequential.py

Um cliente HTTP sequencial, com tratamento de erros adequado e exibição de uma barra de progresso. Sua função *download_one* também é usada por *flags2_threadpool.py*.

flags2_threadpool.py

Um cliente HTTP concorrente baseado em futures. *ThreadPoolExecutor* para mostrar tratamento de erros e integração da barra de progresso.

flags2_asyncio.py

Mesma funcionalidade do exemplo anterior, porém implementada com *asyncio* e *aiohttp*. Será discutido na seção “Melhorando o script para download com *asyncio*” na página 612, no capítulo 18.

**Tome cuidado ao testar clientes concorrentes**

Ao testar clientes HTTP concorrentes em servidores HTTP públicos, você pode gerar muitas requisições por segundo, e é assim que ocorrem os ataques DoS (Denial-of-Services, ou Negação de Serviços). Não queremos atacar ninguém; queremos apenas aprender a criar clientes de alto desempenho. Restrinja cuidadosamente seus clientes quando acessar servidores públicos. Para experimentos com alta concorrência, configure um servidor HTTP local para testes. Instruções para isso estão no arquivo *README.rst* (<http://bit.ly/1Jlsg2L>) no diretório *17-futures/countries/* do repositório de código de *Python fluente* (<http://bit.ly/1JlItSti>).

O recurso mais visível nos exemplos da série *flags2* é a barra de progresso animada, em modo texto, implementada com o pacote *TQDM* (<https://github.com/noamraph/tqdm>). Postei um vídeo de 108s no YouTube (<https://www.youtube.com/watch?v=M8Z65tAl5l4>) para mostrar a barra de progresso e comparar a velocidade dos três scripts *flags2*. No vídeo, começo com o download sequencial, mas interrompo-o após 32s porque demoraria mais de 5 minutos para acessar 676 URLs e obter 194 bandeiras; então executo os scripts com *threads* e com *asyncio*, três vezes cada, e eles sempre terminam a tarefa em 6s ou menos (ou seja, são mais de 60 vezes mais rápidos). A figura 17.1 mostra duas capturas de tela: durante e após a execução de *flags2_threadpool.py*.

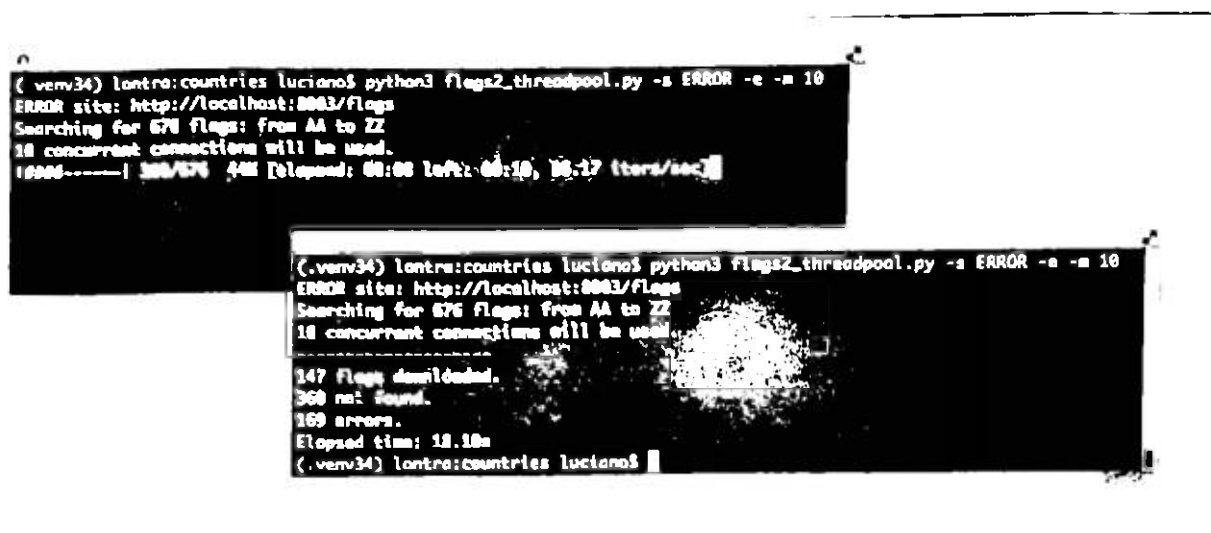


Figura 17.1 – Parte superior à esquerda: `flags2_threadpool.py` executando com uma barra de progresso ativa gerada por `tqdm`; parte inferior à direita: mesma janela de terminal após o término do script.

O pacote TQDM é muito fácil de usar; o exemplo mais simples aparece em um `.gif` animado no `README.md` (<https://github.com/noamraph/tqdm/blob/master/README.md>) do projeto. Se digitar o código a seguir no console de Python após instalar o pacote `tqdm`, você verá uma barra de progresso animada no ponto em que está o comentário:

```
>>> import time
>>> from tqdm import tqdm
>>> for i in tqdm(range(1000)):
...     time.sleep(.01)
...
>>> # -> barra de progresso aparecerá aqui <-
```

Além do belo efeito, a função `tqdm` também é conceitualmente interessante: ela consome qualquer iterável e produz um iterador que, enquanto é consumido, exibe a barra de progresso e estima o tempo restante para concluir todas as iterações. Para calcular essa estimativa, `tqdm` precisa obter um iterável que tenha um `len` ou deve receber o número esperado de itens como segundo argumento. Integrar TQDM com nossos exemplos da série `flags2` oferece uma oportunidade de olhar com mais detalhes o verdadeiro funcionamento dos scripts concorrentes, forçando-nos a usar as funções `futures.as_completed` (<http://bit.ly/1JlsEOW>) e `asyncio.as_completed` (<http://bit.ly/1JluV1>) para que `tqdm` possa exibir o progresso à medida que cada `future` termina.

A outra funcionalidade dos exemplos da série `flags2` é uma interface de linha de comando. Todos os três scripts aceitam as mesmas opções, e você pode vê-las executando qualquer um dos scripts com a opção `-h`. O exemplo 17.8 mostra o texto de ajuda.

Exemplo 17.8 – Tela de ajuda dos scripts da série `flags2`

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT] [-s LABEL]
```

```
[-v]
[CC [CC ...]]
```

Download flags for country codes. Default: top 20 countries by population.

positional arguments:

CC country code or 1st letter (eg. B for BA...BZ)

optional arguments:

```
-h, --help          show this help message and exit
-a, --all           get all available flags (AD to ZW)
-e, --every         get flags for every possible code (AA...ZZ)
-l N, --limit N     limit to N first codes
-m CONCURRENT, --max_req CONCURRENT
                    maximum concurrent requests (default=30)
-s LABEL, --server LABEL
                    Server to hit; one of DELAY, ERROR, LOCAL, REMOTE
                    (default=LOCAL)
-v, --verbose       output detailed progress info
```

Todos os argumentos são opcionais. Os argumentos mais importantes serão discutidos a seguir.

Uma opção que você não pode ignorar é `-s/--server`: ela permite escolher o servidor HTTP e a URL base a serem usados no teste. Você pode passar uma das quatro strings a seguir para determinar em que lugar o script procurará as bandeiras (esses scripts não fazem distinção entre minúsculas e maiúsculas na opção `-s`):

LOCAL

Usa `http://localhost:8001/flags`; é o default. Você deve configurar um servidor HTTP local para responder na porta 8001. Usei Nginx em meus testes. O arquivo *README.rst* (<http://bit.ly/1Jlsg2L>) no diretório de exemplos deste capítulo explica como instalá-lo e configurá-lo.

REMOTE

Usa `http://flupy.org/data/flags`; é um site público meu, hospedado em um servidor compartilhado. Por favor, não o sobrecarregue com muitas requisições concorrentes. O domínio `flupy.org` usa uma conta gratuita na CDN Cloudflare (<http://www.cloudflare.com/>), portanto os primeiros downloads podem ser mais lentos, mas ficarão mais rápidos quando o cache da CDN aquecer.⁶

⁶ Antes de configurar a CDN Cloudflare, obtive erros HTTP 503 – Service Temporarily Unavailable (Serviço temporariamente indisponível) – enquanto testava os scripts com algumas dúzias de requisições concorrentes em minha conta de host compartilhado de baixo custo. Com a CDN esses erros não acontecem mais.

DELAY

Usa `http://localhost:8002/flags`; um proxy que atrasa respostas HTTP deve estar ouvindo na porta 8002. Usei um Mozilla Vaurien na frente de meu Nginx local para introduzir atrasos. O arquivo *README.rst* (<http://bit.ly/1Jlsg2L>) mencionado antes contém instruções para executar um proxy Vaurien.

ERROR

Usa `http://localhost:8003/flags`; um proxy que introduz erros HTTP e atrasos nas respostas deve estar instalado na porta 8003. Usei uma configuração diferente de Vaurien para isso.



A opção LOCAL só funcionará se você configurar e iniciar um servidor HTTP local na porta 8001. As opções DELAY e ERROR exigem proxies ouvindo nas portas 8002 e 8003. A configuração de Nginx e de Mozilla Vaurien para permitir essas opções está explicada no arquivo *17-futures/countries/README.rst* (<http://bit.ly/1Jlsg2L>) no repositório de código de *Python fluente* no GitHub (<https://github.com/fluentpython/example-code>).

Por padrão, cada script da série *flags2* buscará as bandeiras dos vinte países mais populosos no servidor LOCAL (`http://localhost:8001/flags`) usando um número default de conexões concorrentes, que varia de script para script. O exemplo 17.9 tem uma amostra da execução do script *flags2_sequential.py* usando todos os defaults.

Exemplo 17.9 – Executando *flags2_sequential.py* com todos os defaults: site LOCAL, 20 primeiras bandeiras e uma conexão concorrente

```
$ python3 flags2_sequential.py
LOCAL site: http://localhost:8001/flags
Searching for 20 flags: from BD to VN
1 concurrent connection will be used.
.....
20 flags downloaded.
Elapsed time: 0.10s
```

Você pode escolher quais bandeiras serão baixadas de várias maneiras. O exemplo 17.10 mostra como fazer download de todas as bandeiras com códigos de países que comecem com as letras A, B ou C.

Exemplo 17.10 – Executa *flags2_threadpool.py* para buscar todas as bandeiras com códigos de países começados com A, B ou C no servidor DELAY

```
$ python3 flags2_threadpool.py -s DELAY a b c
DELAY site: http://localhost:8002/flags
Searching for 78 flags: from AA to CZ
```

```
30 concurrent connections will be used.
```

```
.....
```

```
43 flags downloaded.
```

```
35 not found.
```

```
Elapsed time: 1.72s
```

Independentemente de como os códigos dos países são selecionados, a quantidade de bandeiras a ser buscada pode ser limitada com a opção `-l/--limit`. O exemplo 17.11 mostra como executar exatamente 100 requisições, combinando a opção `-a` para obter todas as bandeiras com `-l 100`.

Exemplo 17.11 – Executa `flags2_asyncio.py` para obter 100 bandeiras (`-al 100`) do servidor ERROR, usando 100 requisições concorrentes (`-m 100`)

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
```

```
ERROR site: http://localhost:8003/flags
```

```
Searching for 100 flags: from AD to LK
```

```
100 concurrent connections will be used.
```

```
.....
```

```
73 flags downloaded.
```

```
27 errors.
```

```
Elapsed time: 0.64s
```

Essa é a interface de usuário dos exemplos da série `flags2`. Vamos ver como eles são implementados.

Tratamento de erros nos exemplos da série `flags2`

A estratégia comum adotada em todos os três exemplos para lidar com erros HTTP é tratar erros 404 (Not Found, ou Não encontrado) na função responsável pelo download de um único arquivo (`download_one`). Qualquer outra exceção deve ser propagada para ser tratada pela função `download_many`.

Novamente, começaremos estudando o código sequencial, que é mais fácil de acompanhar – e boa parte dele é reutilizada pelo script com pool de threads. O exemplo 17.12 mostra as funções que executam os downloads propriamente ditos nos scripts `flags2_sequential.py` e `flags2_threadpool.py`.

Exemplo 17.12 – `flags2_sequential.py`: funções básicas responsáveis pelo download; ambas são reutilizadas em `flags2_threadpool.py`

```
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = requests.get(url)
```

```

if resp.status_code != 200: ❶
    resp.raise_for_status()
return resp.content

def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc: ❷
        res = exc.response
        if res.status_code == 404:
            status = HTTPStatus.not_found ❸
            msg = 'not found'
        else: ❹
            raise
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
        msg = 'OK'
    if verbose: ❺
        print(cc, msg)

    return Result(status, cc) ❻

```

- ❶ `get_flag` não faz tratamento de erros; usa `requests.Response.raise_for_status` para levantar uma exceção para qualquer código HTTP diferente de 200.
- ❷ `download_one` captura `requests.exceptions.HTTPError` para tratar especificamente o código HTTP 404...
- ❸ ... configurando seu status local para `HTTPStatus.not_found`; `HTTPStatus` é um Enum importado de `flags2_common` (Exemplo A.10).
- ❹ Qualquer outra exceção `HTTPError` é levantada novamente; outras exceções simplesmente se propagarão até quem fez a chamada.
- ❺ Se a opção de linha de comando `-v/--verbose` estiver definida, o código do país e a mensagem de status serão exibidos; é assim que você verá a barra de progresso em modo verboso.
- ❻ A namedtuple `Result` devolvida por `download_one` terá um campo `status` com um valor igual a `HTTPStatus.not_found` ou `HTTPStatus.ok`.

O exemplo 17.13 lista a versão sequencial da função `download_many`. Esse código é simples, mas vale a pena estudá-lo para comparar com as versões concorrentes que estão por vir. Preste atenção no modo como ela informa o progresso, trata erros e contabiliza os downloads.

Exemplo 17.13 – `flags2_sequential.py`: a implementação sequencial de `download_many`

```
def download_many(cc_list, base_url, verbose, max_req):
    counter = collections.Counter() ❶
    cc_iter = sorted(cc_list) ❷
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter) ❸
    for cc in cc_iter: ❹
        try:
            res = download_one(cc, base_url, verbose) ❺
        except requests.exceptions.HTTPError as exc: ❻
            error_msg = 'HTTP error {res.status_code} - {res.reason}'
            error_msg = error_msg.format(res=exc.response)
        except requests.exceptions.ConnectionError as exc: ❼
            error_msg = 'Connection error'
        else: ❽
            error_msg = ''
            status = res.status

        if error_msg:
            status = HTTPStatus.error ❾
        counter[status] += 1 ❿
        if verbose and error_msg: ⓫
            print('*** Error for {}: {}'.format(cc, error_msg))

    return counter ⓬
```

- ❶ Esse Counter contabilizará os diferentes resultados dos downloads: `HTTPStatus.ok`, `HTTPStatus.not_found` ou `HTTPStatus.error`.
- ❷ `cc_iter` armazena a lista de códigos de países recebida como argumento, em ordem alfabética.
- ❸ Se não estiver executando em modo verboso, `cc_iter` é passado para a função `tqdm`; ela devolverá um iterador que produzirá itens em `cc_iter`, ao mesmo tempo que exibe a barra de progresso animada.
- ❹ Esse laço `for` faz uma iteração por `cc_iter` e...
- ❺ ... faz o download por meio de sucessivas chamadas a `download_one`.
- ❻ Exceções relacionadas a HTTP levantadas por `get_flag` e não tratadas por `download_one` são tratadas aqui.
- ❼ Outras exceções relacionadas à rede são tratadas aqui. Qualquer outra exceção abortará o script, pois a função `flags2_common.main` que chama `download_many` não tem `try/except`.

- ❸ Se nenhuma exceção escapar de `download_one`, `status` será recuperado da `namedtuple HTTPStatus` devolvida por `download_one`.
- ❹ Se houve um erro, define o `status` local de acordo com o erro.
- ❺ Incrementa o contador usando o valor do `Enum HTTPStatus` como chave.
- ❻ Se estiver executando em modo verboso, exibe a mensagem de erro para o código do país atual, se houver.
- ❼ Devolve `counter` para que a função `main` possa exibir os números em seu relatório final.

Vamos agora estudar o exemplo com pool de threads refatorado, *flags2_threadpool.py*.

Usando `futures.as_completed`

Para integrar a barra de progresso de TQDM e tratar erros em cada requisição, o script *flags2_threadpool.py* usa `futures.ThreadPoolExecutor` com a função `futures.as_completed`, que já vimos. O exemplo 17.14 mostra a listagem completa de *flags2_threadpool.py*. Somente a função `download_many` é implementada; as demais funções são reutilizadas dos módulos `flags2_common` e `flags2_sequential`

Exemplo 17.14 – *flags2_threadpool.py*: listagem completa

```
import collections
from concurrent import futures

import requests
import tqdm ❶

from flags2_common import main, HTTPStatus ❷
from flags2_sequential import download_one ❸

DEFAULT_CONCUR_REQ = 30 ❹
MAX_CONCUR_REQ = 1000 ❺

def download_many(cc_list, base_url, verbose, concur_req):
    counter = collections.Counter()
    with futures.ThreadPoolExecutor(max_workers=concur_req) as executor: ❶
        to_do_map = {} ❷
        for cc in sorted(cc_list): ❸
            future = executor.submit(download_one,
                                     cc, base_url, verbose) ❹
            to_do_map[future] = cc ❺
        done_iter = futures.as_completed(to_do_map) ❻
```

```

if not verbose:
    done_iter = tqdm.tqdm(done_iter, total=len(cc_list)) ❶
for future in done_iter: ❷
    try:
        res = future.result() ❸
    except requests.exceptions.HTTPError as exc: ❹
        error_msg = 'HTTP {res.status_code} - {res.reason}'
        error_msg = error_msg.format(res=exc.response)
    except requests.exceptions.ConnectionError as exc:
        error_msg = 'Connection error'
    else:
        error_msg = ''
        status = res.status

    if error_msg:
        status = HTTPStatus.error
    counter[status] += 1
    if verbose and error_msg:
        cc = to_do_map[future] ❺
        print('*** Error for {}: {}'.format(cc, error_msg))

return counter

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)

```

- ❶ Importa a biblioteca para exibição da barra de progresso.
- ❷ Importa uma função e um Enum do módulo `flags2_common`.
- ❸ Reutiliza `download_one` de `flags2_sequential` (Exemplo 17.12).
- ❹ Se a opção de linha de comando `-m/--max_req` não for especificada, esse será o número máximo de requisições concorrentes, implementado como o tamanho do pool de threads; o número final poderá ser menor se o número de bandeiras a ser baixado for menor.
- ❺ `MAX_CONCUR_REQ` limita o número máximo de requisições concorrentes, independentemente do número de bandeiras a serem baixadas ou da opção de linha de comando `-m/--max_req`; é um dispositivo de proteção.
- ❻ Cria o executor com `max_workers` definido com `concur_req`, calculado pela função `main` como o menor valor entre `MAX_CONCUR_REQ`, o tamanho de `cc_list` e o valor da opção de linha de comando `-m/--max_req`. Isso evita criar mais threads que o necessário.
- ❼ Esse dict mapeará cada instância de `Future` – representando um download – com o respectivo código do país para informar erros.

- ❸ Itera pela lista de códigos de países em ordem alfabética. A ordem dos resultados dependerá, acima de tudo, do tempo de cada resposta HTTP, mas se o tamanho do pool de threads (dado por `concur_req`) for muito menor que `len(cc_list)`, você poderá perceber os downloads agrupados em ordem alfabética.
- ❹ Cada chamada a `executor.submit` agenda a execução de um invocável e devolve uma instância de `future`. O primeiro argumento é o invocável; o restante são os argumentos que ele receberá.
- ❺ Armazena o `future` e o código do país em um `dict`.
- ❻ `futures.as_completed` devolve um iterador que produz `futures` à medida que eles terminam.
- ❼ Se não estiver em modo verboso, encapsula o resultado de `as_completed` com a função `tqdm` para exibir a barra de progresso; como `done_iter` não tem `len`, precisamos informar o número esperado de itens à `tqdm` com o argumento `total=` para que `tqdm` possa estimar o trabalho restante.
- ❽ Itera pelos `futures` à medida que estes terminam.
- ❾ Chamar o método `result` em um `future` devolve o valor retornado pelo invocável ou levanta qualquer exceção capturada quando o invocável foi executado. Esse método pode ficar bloqueado à espera de uma resposta, mas isso não ocorre nesse exemplo porque `as_completed` devolve somente os `futures` terminados.
- ❿ Trata as exceções em potencial; o restante dessa função é idêntico à versão sequencial de `download_many` (Exemplo 17.13), exceto pela próxima linha numerada para comentário.
- ⓫ Para ter um contexto para a mensagem de erro, recupera o código do país de `to_do_map` usando o `future` atual como chave. Isso não era necessário na versão sequencial porque estávamos iterando pela lista de códigos de países, portanto tínhamos o `cc` atual; nesse caso, estamos iterando pelos `futures`.

O exemplo 17.14 usa uma técnica (idiom) muito útil com `futures.as_completed`: criar um `dict` para mapear cada `future` a outros dados que possam ser úteis quando o `future` terminar. Nesse caso, `to_do_map` mapeia cada `future` ao código do país atribuído a ele. Isso facilita o processamento posterior do resultado dos `futures`, já que eles são produzidos fora de ordem.

Threads em Python são muito convenientes para aplicações intensivas em E/S com a rede, e o pacote `concurrent.futures` as deixam muito simples para utilização em determinados casos de uso. Com isso, concluímos nossa introdução básica a `concurrent.futures`. Vamos agora discutir alternativas para as ocasiões em que `ThreadPoolExecutor` ou `ProcessPoolExecutor` não forem adequados.

Alternativas com threading e multiprocessing

Python tem suporte para threads desde sua versão 0.9.8 (1993); `concurrent.futures` é apenas a maneira mais recente de usá-las. Em Python 3, o módulo `thread` original tornou-se obsoleto em favor do módulo `threading` (<https://docs.python.org/3/library/threading.html>) de mais alto nível.⁷ Se `futures.ThreadPoolExecutor` não for suficientemente flexível para uma determinada tarefa, talvez você precise desenvolver sua própria solução a partir dos componentes básicos de `threading`, como `Thread`, `Lock`, `Semaphore` etc. – possivelmente usando as filas seguras para threads do módulo `queue` (<https://docs.python.org/3/library/queue.html>) para trocar dados entre threads. Esses componentes estão embutidos em `futures.ThreadPoolExecutor`.

Para tarefas limitadas por CPU, precisamos evitar a GIL iniciando vários processos. `futures.ProcessPoolExecutor` é a maneira mais fácil de fazer isso. Mas, novamente, se seu caso de uso for complexo, você precisará de ferramentas mais sofisticadas. O pacote `multiprocessing` (<https://docs.python.org/3/library/multiprocessing.html>) emula a API de `threading`, mas delega tarefas a vários processos. Em programas simples, `multiprocessing` pode substituir `threading` com poucas alterações. Porém `multiprocessing` também oferece recursos para solucionar o maior desafio de processos cooperativos: trocar dados.

Resumo do capítulo

Iniciamos o capítulo comparando dois clientes HTTP concorrentes com um cliente sequencial, mostrando que há ganhos significativos de desempenho em relação ao script sequencial.

Depois de estudar o primeiro exemplo baseado em `concurrent.futures`, demos uma olhada mais de perto em objetos futures, sejam instâncias de `concurrent.futures.Future` ou de `asyncio.Future`, destacando o que essas classes têm em comum (suas diferenças serão enfatizadas no capítulo 18). Vimos como criar futures chamando `Executor.submit()` e iteramos por futures já executados com `concurrent.futures.as_completed()`.

Em seguida, vimos por que threads em Python não funcionam bem em aplicações limitadas por E/S (I/O-bound), apesar da GIL: toda função de E/S da biblioteca-padrão escrita em C libera a GIL, portanto, enquanto uma dada thread espera por E/S, o escalonador de Python pode alternar para outra thread. Em seguida, discutimos o uso de vários processos com a classe `concurrent.futures.ProcessPoolExecutor` para contornar a GIL e usar vários núcleos de CPU para executar algoritmos de

⁷ O módulo `threading` está disponível desde Python 1.5.1 (1998), embora algumas pessoas insistam em usar o velho módulo `thread`. Em Python 3, foi renomeado para `_thread` para enfatizar que é somente um detalhe de implementação de baixo nível, e não deve ser usado em código de aplicação.

criptografia, conseguindo ganhos de velocidade de mais de 100% quando usamos quatro processos de trabalho.

Na seção seguinte, vimos de perto como `concurrent.futures.ThreadPoolExecutor` funciona, com um exemplo didático iniciando tarefas que não fazem nada por alguns segundos, exceto exibir seus status com um timestamp.

Em seguida, retomamos os exemplos de download de bandeiras. Melhorá-los com uma barra de progresso e com um tratamento de erros apropriado possibilitou uma melhor exploração da função geradora `future.as_completed`, mostrando um padrão comum: armazenar futures em um dict para associar outras informações a eles na submissão, para podermos usar essas informações quando o future for produzido pelo iterador `as_completed`.

Concluimos a discussão sobre concorrência com threads e processos com uma rápida menção aos módulos `threading` e `multiprocessing` de baixo nível, porém mais flexíveis, que são a forma tradicional de aproveitar threads e processos em Python.

Leituras complementares

O pacote `concurrent.futures` foi uma contribuição de Brian Quinlan, que o apresentou em uma ótima palestra chamada “The Future Is Soon!” (O futuro está logo ali!, <http://bit.ly/1JluZJy>) na PyCon Austrália em 2010. A palestra de Quinlan não tem slides; ele mostra o que a biblioteca faz digitando código diretamente no console de Python. Como exemplo para motivação, a apresentação tem um pequeno vídeo com o cartunista/programador Randall Munroe do XKCD fazendo um ataque DoS não intencional ao Google Maps para criar um mapa colorido dos tempos de percursos pela sua cidade. A introdução formal à biblioteca está na *PEP 3148 – futures – execute computations asynchronously* (*futures – execute processamentos assincronamente*, <https://www.python.org/dev/peps/pep-3148/>). Na PEP, Quinlan escreveu que a biblioteca `concurrent.futures` foi “altamente influenciada pelo pacote `java.util.concurrent` de Java”.

O livro *Parallel Programming with Python* (Packt) de Jan Palach discute várias ferramentas para programação concorrente, incluindo os módulos `concurrent.futures`, `threading` e `multiprocessing`. Ele vai além da biblioteca-padrão para discutir Celery (<http://bit.ly/1Jlv1kA>), um gerenciador de filas de tarefas usado para distribuir trabalhos entre threads e processos, mesmo em computadores diferentes. Na comunidade Django, Celery provavelmente é o sistema mais amplamente usado para delegar tarefas pesadas, como a geração de PDFs, a outros processos, evitando assim atrasos na geração de uma resposta HTTP.

No livro *Python Cookbook*, 3E (O'Reilly, <http://shop.oreilly.com/product/0636920027072.do>)⁸ de Beazley e Jones, há receitas que usam `concurrent.futures`, começando pela “Recipe 11.12. Understanding Event-Driven I/O” (Entender como funciona o I/O orientado a eventos). A “Recipe 12.7. Creating a Thread Pool” (Criar um pool de threads) mostra um servidor TCP de eco simples, e a “Recipe 12.8. Performing Simple Parallel Programming” (Realizar programação paralela simples) apresenta um exemplo bem prático: analisar todo um diretório de arquivos de log do Apache compactados com `gzip` com a ajuda de um `ProcessPoolExecutor`. Para mais informações sobre threads, todo o capítulo 12 de Beazley e Jones é ótimo, com destaque para a “Recipe 12.10. Defining an Actor Task” (Definir uma tarefa do tipo ator), que mostra o modelo Actor: um modo elegante e comprovado de coordenar threads por meio de troca de mensagens.

O livro *Effective Python* (<http://www.effectivepython.com/>) de Brett Slatkin tem um capítulo com assuntos variados sobre concorrência, incluindo corrotinas, `concurrent.futures` com threads e processos, bem como o uso de travas (locks) e filas para programação com threads sem `ThreadPoolExecutor`.

Os livros *High Performance Python* (O'Reilly, <http://shop.oreilly.com/product/0636920028963.do>) de Micha Gorelick e Ian Ozsvald e *The Python Standard Library by Example* (Addison-Wesley) de Doug Hellmann também discutem threads e processos.

Para uma abordagem moderna sobre concorrência sem threads ou callbacks, o livro *Seven Concurrency Models in Seven Weeks* de Paul Butcher (Pragmatic Bookshelf) é uma excelente leitura. Adoro o seu subtítulo: “When Threads Unravel”⁹. Nesse livro, threads e travas são discutidas no capítulo 1, e os seis capítulos restantes são dedicados a alternativas modernas à programação concorrente, conforme implementadas em diferentes linguagens. Python, Ruby e JavaScript não estão entre elas.

Se estiver curioso sobre a GIL, comece em *Python Library and Extension FAQ* (Perguntas frequentes sobre bibliotecas e extensões de Python) com a pergunta “Can't we get rid of the Global Interpreter Lock?” (Não podemos nos livrar da Trava Global do Interpretador?, <http://bit.ly/1HGtb0F>). Também vale a pena ler os posts de Guido van Rossum e Jesse Noller (contribuidor do pacote `multiprocessing`): “It isn't Easy to Remove the GIL” (Não é fácil remover a GIL, <http://bit.ly/1HGtcBF>) e “Python Threads and the Global Interpreter Lock” (Threads em Python e a Trava Global do Interpretador, <http://bit.ly/1Jlvwd>). Por fim, David Beazley faz uma exploração detalhada do funcionamento interno da GIL: “Understanding the Python GIL” (Entendendo a GIL de Python, <http://www.dabeaz.com/GIL/>).¹⁰ No slide 54 da apresentação (<http://bit.ly/1HGtCrK>),

8 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

9 N.T.: Literalmente, pode ser traduzido como “Quando threads desfilam” – um trocadilho, já que “thread” em inglês também pode significar fios de linha. O verbo “to unravel” também pode significar “desvendar”.

10 Agradeço a Lucas Brunialti por ter me enviado um link para essa palestra.

Beazley mostra alguns resultados alarmantes, incluindo um aumento de vinte vezes no tempo de processamento em um benchmark em particular com o novo algoritmo da GIL introduzido em Python 3.2. No entanto Beazley aparentemente usou um `while True: pass` vazio para simular uma tarefa limitada por CPU, e isso não é realista. O problema não é significativo em cargas de trabalho reais, de acordo com um comentário de Antoine Pitrou (<http://bugs.python.org/issue7946#msg223110>) – que implementou o novo algoritmo da GIL – no relatório de bug submetido por Beazley.

Como a GIL é um problema real e é provável que não desapareça tão cedo, Jesse Noller e Richard Oudkerk colaboraram com uma biblioteca para contornar mais facilmente a GIL em aplicações limitadas por CPU: o pacote `multiprocessing`, que emula a API de `threading` para processos, além de oferecer uma infraestrutura de suporte para travas, filas, pipes, memória compartilhada etc. O pacote foi introduzido na *PEP 371 – Addition of the multiprocessing package to the standard library* (Adição do pacote `multiprocessing` à biblioteca-padrão, <https://www.python.org/dev/peps/pep-0371/>). A documentação oficial do pacote (<http://bit.ly/multi-docs>) é um arquivo `.rst` de 93 KB – são aproximadamente 63 páginas –, o que faz dele um dos capítulos mais longos da biblioteca-padrão de Python. `multiprocessing` é a base de `concurrent.futures.ProcessPoolExecutor`.

Para processamentos paralelos com uso intensivo de CPU e grandes volumes de dados, uma nova opção que está ganhando bastante impulso na comunidade de big data é o sistema de processamento distribuído Apache Spark (<https://spark.apache.org/>), que oferece uma API amigável para Python e suporte a objetos Python como dados, conforme mostra sua página de exemplos (<https://spark.apache.org/examples.html>).

Duas bibliotecas elegantes e muito fáceis de usar para paralelizar tarefas com processos são `lelo` (<https://pypi.python.org/pypi/lelo>) de João S. O. Bueno e `python-parallelize` (<http://bit.ly/1HGtF6Q>) de Nat Pryce. O pacote `lelo` define um decorador `@parallel` que você pode aplicar a qualquer função para deixá-la não bloqueante em um passe de mágica: ao chamar a função decorada, sua execução será iniciada em outro processo. O pacote `python-parallelize` de Nat Pryce oferece um gerador `parallelize` que você pode usar para distribuir a execução de um laço `for` em várias CPUs. Ambos pacotes usam o módulo `multiprocessing` internamente.

Ponto de vista

Evitando threads

Concorrência: um dos assuntos mais difíceis em ciência da computação (normalmente é melhor evitá-lo).¹¹

— David Beazley

Instrutor de Python e cientista maluco

Concordo com a aparente contradição entre esta citação de David Beazley e de Michele Simionato no início deste capítulo. Após ter participado de um curso sobre concorrência na universidade – em que “programação concorrente” era sinônimo de administrar threads e travas –, cheguei à conclusão de que não quero administrar threads e travas por conta própria, assim como não gosto de administrar alocação e liberação de memória. Essas tarefas são mais bem executadas por programadores de sistemas com o know-how, a inclinação e o tempo para fazê-las corretamente – ou assim esperamos.

É por isso que acho o pacote `concurrent.futures` empolgante: ele trata threads, processos e filas como uma infraestrutura a seu dispor, e não como algo com que você precise lidar diretamente. É claro que ele foi projetado com tarefas simples em mente, ou seja, os chamados problemas “embaraçosamente paralelos” (*embarrassingly parallel*, <http://bit.ly/1HGtGaR>). Mas essa é uma grande parcela dos problemas de concorrência com que nos deparamos quando escrevemos aplicações – e não sistemas operacionais ou servidores de banco de dados, como destaca Simionato em sua citação.

Para problemas de concorrência “não embaraçosos”, threads e travas tampouco são a resposta. As threads jamais desaparecerão no nível do sistema operacional, mas todas as linguagens de programação que achei interessantes nos últimos anos oferecem abstrações melhores e de mais alto nível para concorrência, como mostra o livro *Seven Concurrency Models*. Go, Elixir e Clojure estão entre elas. Erlang – a linguagem de implementação de Elixir – é um excelente exemplo de uma linguagem projetada desde o princípio com concorrência em mente. Não curti Erlang por um único motivo: achei sua sintaxe feia. Python me deixou mal acostumado nesse quesito.

¹¹ Slide 9 do tutorial “A Curious Course on Coroutines and Concurrency” (Um curioso curso sobre corrotinas e concorrência, <http://www.dabeaz.com/coroutines/>) apresentado na PyCon 2009.

José Valim, conhecido como colaborador do núcleo de Ruby on Rails, projetou Elixir com uma sintaxe agradável e moderna. Como Lisp e Clojure, Elixir implementa macros sintáticas. É uma faca de dois gumes. Macros sintáticas permitem criar DSLs poderosas, mas a proliferação de sublinguagens pode levar a bases de código incompatíveis e à fragmentação da comunidade. Lisp afogou-se em uma enxurrada de macros, com cada variante de Lisp usando seu próprio dialeto arcano. A padronização em torno de Common Lisp resultou em uma linguagem inchada. Espero que José Valim possa inspirar a comunidade Elixir para evitar um resultado parecido.

Como Elixir, Go é uma linguagem moderna, com ideias novas. Porém, em alguns aspectos, é uma linguagem conservadora se comparada com Elixir. Go não tem macros, e sua sintaxe é mais simples que a sintaxe de Python. Go não tem suporte para herança nem sobrecarga de operadores, e oferece menos oportunidades para metaprogramação que Python. Essas limitações são consideradas vantagens da linguagem. Elas resultam em comportamentos e desempenho mais previsíveis. São características desejáveis em ambientes de missão crítica, com alta concorrência, onde Go pretende substituir C++, Java e Python.

Enquanto Elixir e Go são concorrentes diretos nos domínios da alta concorrência, suas filosofias de design atraem públicos-alvos diferentes. É provável que ambas prosperem. Entretanto, na história das linguagens de programação, as linguagens conservadoras tendem a atrair mais programadores. Quero ficar fluente em Go e em Elixir.

Sobre a GIL

A GIL simplifica a implementação do interpretador CPython e de extensões escritas em C, portanto podemos agradecer-lhe pelo grande número de extensões em C disponíveis para Python – e, certamente, esse é um dos motivos principais para Python ser tão popular atualmente.

Por muitos anos, tive a impressão de que a GIL deixava as threads em Python quase inúteis para aplicações que não fossem muito simples. Foi só quando descobri que *toda* chamada de E/S bloqueante da biblioteca-padrão libera a GIL que percebi que as threads em Python são excelentes para sistemas limitados por E/S – o tipo de aplicação que os clientes geralmente me pagam para desenvolver, dada a minha experiência profissional.

Concorrência na concorrência

A MRI – a implementação de referência de Ruby – também tem uma GIL, portanto suas threads estão sujeitas às mesmas limitações de Python. Por outro lado, interpretadores JavaScript não permitem que o usuário crie threads diretamente; a programação assíncrona com callbacks é a única opção para implementar concorrência. Menciono isso porque Ruby e JavaScript são as concorrentes mais diretas de Python como linguagens de programação dinâmicas de propósito geral. Observando a nova geração de linguagens especializadas em concorrência, Go e Elixir provavelmente são as mais bem posicionadas para tomar o lugar de Python. Mas agora temos `asyncio`. Se uma multidão de pessoas acredita que Node.js com suas primitivas callbacks é uma plataforma viável para programação concorrente, o quão difícil será convencê-las a mudar para Python quando o ecossistema em torno de `asyncio` amadurecer? Mas esse é um assunto para a próxima seção “Ponto de vista” na página 640.

CAPÍTULO 18

Concorrência com asyncio

Concorrência diz respeito a lidar com muitas coisas ao mesmo tempo.

Paralelismo diz respeito a fazer muitas coisas ao mesmo tempo.

Não são conceitos iguais, mas estão relacionados.

Um tem a ver com estrutura; o outro, com execução.

Concorrência oferece uma maneira de estruturar uma solução para resolver um problema que pode (mas não necessariamente) ser paralelizável.¹

— Rob Pike
Coinventor da linguagem Go

O professor Imre Simon² gostava de dizer que existem dois grandes pecados nas ciências: usar palavras diferentes para significar o mesmo e usar uma só palavra para significar coisas diferentes. Se fizer uma pesquisa sobre programação concorrente ou paralela, você encontrará diferentes definições para “concorrência” e “paralelismo”. Adotarei as definições informais de Rob Pike, autor da citação anterior.

Para obter um verdadeiro paralelismo, você precisa ter vários cores (núcleos) de CPU. Um notebook moderno tem quatro núcleos de CPU, mas normalmente está rodando mais de cem processos a qualquer momento, em uso normal e cotidiano. Portanto, na prática, a maior parte do processamento ocorre de forma concorrente, e não paralelamente. O computador lida constantemente com mais de cem processos, garantindo que cada um tenha a oportunidade de progredir, mesmo que a CPU não consiga fazer mais de quatro tarefas ao mesmo tempo. Dez anos atrás, usávamos computadores que eram também capazes de tratar cem processos de forma concorrente, mas com um

1 Slide 5 da palestra “Concurrency Is Not Parallelism (It’s Better)” [Concorrência não é paralelismo (é melhor), <http://bit.ly/1OwVTUf>].

2 Imre Simon (1943–2009) foi pioneiro em ciência da computação no Brasil, fez contribuições essenciais à Teoria de Autômatos, e deu início ao campo da Matemática Tropical. Também era defensor de software livre e cultura livre. Tive a felicidade de estudar, trabalhar e conviver com ele.

único núcleo. É por isso que Rob Pike deu à palestra o nome “Concurrency Is Not Parallelism (It’s Better)” [Concorrência não é paralelismo (é melhor)].

Este capítulo apresenta o pacote `asyncio`, que implementa concorrência com corrotinas acionadas por um loop de eventos. É uma das maiores e mais ambiciosas bibliotecas já adicionadas ao Python. Guido van Rossum desenvolveu o `asyncio` fora do repositório de Python e deu ao projeto o codinome “Tulip” – portanto você verá referências a essa flor (tulipa) quando pesquisar esse assunto online. Por exemplo, o principal grupo de discussão sobre este tema até hoje se chama `python-tulip` (<http://bit.ly/1HGtMiO>).

Tulip passou a se chamar `asyncio` quando foi adicionado à biblioteca-padrão em Python 3.4. Ele também é compatível com Python 3.3 – você pode encontrá-lo no PyPI com o novo nome oficial (<https://pypi.python.org/pypi/asyncio>). Pelo fato de usar expressões `yield from`, `asyncio` é incompatível com versões mais antigas de Python.



O projeto Trollius (<http://trollius.readthedocs.org/>) – que também tem nome de flor – é um porte de `asyncio` para Python 2.6 e versões mais recentes, substituindo `yield from` por `yield` e invocáveis com nomes espertos como `From` e `Return`. Uma expressão `yield from _` torna-se `yield From(_)`; quando uma corrotina precisa devolver um resultado, você escreve `raise Return(result)` em vez de `return result`. O projeto Trollius é liderado por Victor Stinner, que é também um core developer de `asyncio`; ele gentilmente concordou em revisar este capítulo na produção deste livro.

Neste capítulo, você verá:

- Uma comparação entre um programa simples com threads e o equivalente com `asyncio`, mostrando a relação entre threads e tarefas assíncronas.
- Como a classe `asyncio.Future` difere de `concurrent.futures.Future`.
- Versões assíncronas dos exemplos de download de bandeiras do capítulo 17.
- Como a programação assíncrona consegue obter alta concorrência em aplicações de rede sem exigir threads ou processos.
- Como as corrotinas são uma grande melhoria em relação a callbacks para programação assíncrona.
- Como evitar o bloqueio do loop de eventos passando operações bloqueantes para um pool de threads.
- Como escrever servidores com `asyncio` e repensar as aplicações web para que consigam lidar com alta concorrência.
- Por que `asyncio` está prestes a causar um enorme impacto no ecossistema de Python.

Vamos começar com o exemplo simples que compara `threading` e `asyncio`.

Thread versus corrotina: uma comparação

Durante uma discussão sobre threads e a GIL, Michele Simionato postou um exemplo simples, porém divertido (<http://bit.ly/1Ox3vWA>), usando multiprocessing para exibir uma barra giratória animada (o spinner), formada pela alternância dos caracteres ASCII "|/-\" no console enquanto um processamento demorado estava executando.

Adapte o exemplo de Simionato para usar uma thread com o módulo Threading e, então, uma corrotina com asyncio, para que você possa ver os dois exemplos lado a lado e entender como implementar comportamentos concorrentes sem threads.

A saída mostrada nos exemplos 18.1 e 18.2 é animada, portanto você precisa realmente executar os scripts para ver o que acontece. Se estiver no metrô (ou em outro lugar sem conexão WiFi), dê uma olhada na figura 18.1 e imagine que a barra (\) antes da palavra “thinking” está girando.

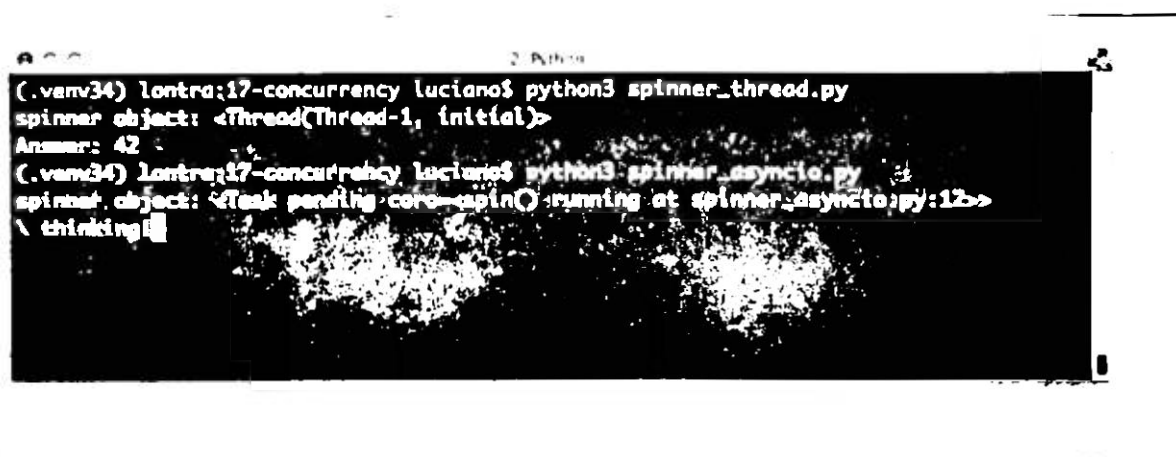


Figura 18.1 – Os scripts `spinner_thread.py` e `spinner_asyncio.py` geram saídas semelhantes: a repr de um objeto spinner e o texto `Answer: 42`. Na captura de tela, `spinner_asyncio.py` ainda está executando e a mensagem `\ thinking!` do spinner é mostrada; quando o script termina, essa linha é substituída por `Answer: 42`.

Vamos primeiro analisar o script `spinner_thread.py` (Exemplo 18.1).

Exemplo 18.1 – `spinner_thread.py`: animando um spinner textual com uma thread

```
import threading
import itertools
import time
import sys
```

```
class Signal: ❶
    go = True
```

Capítulo 18 ■ Concorrência com `asyncio`

```

def spin(msg, signal): ❹
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'): ❶
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status)) ❷
        time.sleep(.1)
        if not signal.go: ❸
            break
    write(' ' * len(status) + '\x08' * len(status)) ❸

def slow_function(): ❹
    # finge que espera bastante tempo para E/S
    time.sleep(3) ❺
    return 42

def supervisor(): ❶
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner) ❷
    spinner.start() ❸
    result = slow_function() ❹
    signal.go = False ❺
    spinner.join() ❻
    return result

def main():
    result = supervisor() ❻
    print('Answer:', result)

if __name__ == '__main__':
    main()

```

- ❶ Essa classe define um objeto mutável simples com um atributo `go` que usaremos para controlar a thread de fora.
- ❷ Essa função rodará em uma thread separada. O argumento `signal` é uma instância da classe `Signal` que acabou de ser definida.
- ❸ Esse, na verdade, é um laço infinito, pois `itertools.cycle` produz itens percorrendo repetidamente a sequência fornecida.

- ④ O truque para fazer animação em modo texto: mover o cursor para trás com caracteres de backspace (`\x08`).
- ⑤ Se o atributo `go` não for mais `True`, sai do laço.
- ⑥ Limpa a linha de status sobrescrevendo com espaços e movendo o cursor de volta ao início.
- ⑦ Imagine que esse seja um processamento de E/S de alta latência.
- ⑧ Chamar `sleep` fará a thread principal ser bloqueada, porém o mais importante é que a GIL será liberada, portanto a thread secundária poderá prosseguir.
- ⑨ Essa função cria a thread secundária, exibe o objeto thread, executa o processamento lento e mata a thread.
- ⑩ Exibe o objeto da thread secundária. A saída é semelhante a `<Thread(Thread-1, initial)>`.
- ⑪ Inicia a thread secundária.
- ⑫ Executa `slow_function`; isso bloqueia a thread principal. Enquanto isso, o spinner é animado pela thread secundária.
- ⑬ Muda o estado de `signal`; isso encerrará o laço `for` da função `spin`.
- ⑭ Espera até a thread `spinner` terminar.
- ⑮ Executa a função `supervisor`.

Observe que, por design, não existe uma API para terminar uma thread em Python. É preciso enviar-lhe uma mensagem para que ela termine por conta própria. Nesse caso, usei o atributo `signal.go`: quando a thread lhe atribui um valor falso, a thread `spinner` em algum momento percebe isso e sai de forma limpa.

Vamos agora ver como o mesmo comportamento pode ser conseguido com `@asyncio.coroutine` em vez de usar uma thread.



Conforme observado na seção “Resumo do capítulo” na página 553 (Capítulo 16), `asyncio` usa uma definição mais rigorosa de “corrotina”. Uma corrotina própria para uso com a API de `asyncio` precisa usar `yield from`, e não `yield` em seu corpo. Além disso, uma corrotina para `asyncio` precisa ser acionada por um chamador (caller) usando `yield from` ou passando a corrotina para uma das funções de `asyncio`, por exemplo, `asyncio.async(_)` e outras discutidas neste capítulo. Por fim, o decorador `@asyncio.coroutine` deve ser aplicado às corrotinas, como mostram os exemplos.

Dê uma olhada no exemplo 18.2.

Exemplo 18.2 – spinner_asyncio.py: animando um spinner textual com uma corrotina

```
import asyncio
import itertools
import sys

@asyncio.coroutine ❶
def spin(msg): ❷
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            yield from asyncio.sleep(.1) ❸
        except asyncio.CancelledError: ❹
            break
    write(' ' * len(status) + '\x08' * len(status))

@asyncio.coroutine
def slow_function(): ❺
    # finge que espera bastante tempo para E/S
    yield from asyncio.sleep(3) ❻
    return 42

@asyncio.coroutine
def supervisor(): ❼
    spinner = asyncio.async(spin('thinking!')) ❽
    print('spinner object:', spinner) ❾
    result = yield from slow_function() ❿
    spinner.cancel() ⓫
    return result

def main():
    loop = asyncio.get_event_loop() ⓫
    result = loop.run_until_complete(supervisor()) ⓫
    loop.close()
    print('Answer:', result)

if __name__ == '__main__':
    main()
```

- ❶ Corrotinas destinadas a serem usadas com `asyncio` devem ser decoradas com `@asyncio.coroutine`. Isso não é obrigatório, mas é bastante aconselhável. Veja a explicação após essa listagem.
- ❷ Nesse caso, não precisamos do argumento `signal` que foi usado para terminar a thread na função `spin` do exemplo 18.1.
- ❸ Usa `yield from asyncio.sleep(.1)` no lugar de apenas `time.sleep(.1)` para dormir sem bloquear o loop de eventos.
- ❹ Se `asyncio.CancelledError` for levantada quando `spin` acordar, é sinal de que seu cancelamento foi solicitado, portanto sairá do laço.
- ❺ `slow_function` agora é uma corrotina e usa `yield from` para deixar o loop de eventos prosseguir enquanto essa corrotina finge estar fazendo E/S, mas na verdade apenas dorme.
- ❻ A expressão `yield from asyncio.sleep(3)` passa o fluxo de controle para o laço principal, que reativará essa corrotina após o tempo decorrido em `sleep`.
- ❼ `supervisor` agora é uma corrotina também, portanto pode acionar `slow_function` com `yield from`.
- ❽ `asyncio.async(...)` escalona a corrotina `spin` para executar, encapsulando-a em um objeto `Task`, que é devolvido imediatamente.³
- ❾ Exibe o objeto `Task`. A saída é semelhante a `<Task pending coro=<spin() running at spinner_asyncio.py:12>>`.
- ❿ Aciona `slow_function()`. Quando ela terminar, obtém o valor devolvido. Enquanto isso, o loop de eventos continuará a executar porque, em última instância, `slow_function` usa `yield from asyncio.sleep(3)` para devolver o controle ao laço principal.
- ⓫ Um objeto `Task` pode ser cancelado: isso levanta `asyncio.CancelledError` na linha com `yield`, no ponto em que a corrotina está suspensa no momento. A corrotina pode capturar a exceção e atrasar ou até mesmo recusar o cancelamento.
- ⓬ Obtém uma referência ao loop de eventos.
- ⓭ Aciona a corrotina `supervisor` até terminar; o valor de retorno da corrotina é o valor de retorno dessa chamada.



Nunca use `time.sleep(...)` em corrotinas com `asyncio`, a menos que você queira bloquear a thread principal, congelando assim o loop de eventos e, provavelmente, toda a aplicação também. Se uma corrotina precisa passar um tempo sem fazer nada, ela deve usar `yield from asyncio.sleep(DELAY)`.

3 Nota do autor/revisor técnico: Depois que o original deste livro ficou pronto, a função `@asyncio.async` foi deprecated. A partir do Python 3.4.4, deve-se usar `@asyncio.ensure_future`, pois em Python 3.5 `async` está em vias de se tornar uma palavra reservada, e já é usada como parte da sintaxe das novas instruções `async def`, `async for` e `async with`.

O uso do decorador `@asyncio.coroutine` não é obrigatório, mas é altamente recomendável: ele faz as corrotinas se destacarem entre as funções comuns e ajuda na depuração gerando um aviso quando uma corrotina é eliminada pelo coletor de lixo sem ter sido acionada – o que significa que alguma operação não foi concluída e, provavelmente, é um bug. Esse não é um *decorador de preparação* (priming decorator).

Observe que o número de linhas de `spinner_thread.py` e de `spinner_asyncio.py` é quase o mesmo. As funções `supervisor` são o coração desses exemplos. Vamos compará-las em detalhes. O exemplo 18.3 mostra apenas o `supervisor` do exemplo com `Threading`.

Exemplo 18.3 – `spinner_thread.py`: a função `supervisor` com `thread`

```
def supervisor():
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner)
    spinner.start()
    result = slow_function()
    signal.go = False
    spinner.join()
    return result
```

Para comparar, o exemplo 18.4 mostra a corrotina `supervisor`.

Exemplo 18.4 – `spinner_asyncio.py`: a corrotina `supervisor` assíncrona

```
@asyncio.coroutine
def supervisor():
    spinner = asyncio.async(spin('thinking!'))
    print('spinner object:', spinner)
    result = yield from slow_function()
    spinner.cancel()
    return result
```

Veja um resumo das principais diferenças a serem notadas entre as duas implementações de `supervisor`:

- Uma `asyncio.Task`, grosso modo, equivale a uma `threading.Thread`. Victor Stinner, revisor técnico especial deste capítulo, destaca que “uma `Task` é como uma `thread` verde (green thread) em bibliotecas que implementam multitarefa cooperativa, como `gevent`”.
- Uma `Task` aciona uma corrotina, enquanto uma `Thread` chama um invocável.

- Você não instancia objetos `Task` diretamente: eles são obtidos passando uma corrotina para `asyncio.async(...)` ou para `loop.create_task(...)`.
- Quando você obtém um objeto `Task`, ele já está escalonado para executar (por exemplo, por `asyncio.async`); uma instância de `Thread` deve ser explicitamente comandada para executar, por meio de seu método `start`.
- No supervisor com `thread`, `slow_function` é uma função comum, chamada diretamente pela `thread`. No supervisor com `asyncio`, `slow_function` é uma corrotina acionada por `yield from`.
- Não existe API para terminar uma `thread` de fora, pois uma `thread` pode ser interrompida em qualquer ponto, deixando o sistema em um estado inválido. Para `tasks`, temos o método de instância `Task.cancel()`, que levanta `CancelledError` na corrotina. A corrotina pode lidar com isso capturando a exceção no `yield` onde estava suspensa.
- A corrotina supervisor deve ser acionada com `loop.run_until_complete` na função `main`.

Essa comparação deve ajudar a entender como tarefas concorrentes são orquestradas com `asyncio`, em comparação com o modo como isso é feito com o familiar módulo `Threading`.

Uma última questão relacionada a *threads versus* corrotinas: se você já fez alguma programação não trivial com `threads`, sabe como é desafiador raciocinar sobre o programa, pois o escalonador pode interromper uma `thread` a qualquer momento. Você precisa se lembrar de controlar travas (`locks`) para proteger seções críticas de seu programa a fim de evitar que um algoritmo seja interrompido no meio de uma operação de vários passos – o que poderia deixar seus dados em um estado inválido.

Com corrotinas, tudo está protegido contra interrupção por padrão. Você precisa ceder o controle explicitamente para deixar o restante do programa executar. Em vez de usar travas para sincronizar as operações de várias `threads`, você tem corrotinas que são “sincronizadas” (`synchronized`) por definição: somente uma delas executa em determinado instante. Quando quiser ceder o controle, use `yield` ou `yield from` para devolvê-lo ao escalonador. É por isso que é possível cancelar uma corrotina de forma segura: por definição, uma corrotina só pode ser cancelada quando está suspensa em um `yield`, de modo que você pode encerrar de forma limpa ao tratar a exceção `CancelledError`.

Veremos agora como a classe `asyncio.Future` difere da classe `concurrent.futures.Future` que vimos no capítulo 17.

`asyncio.Future`: não bloqueante por design

As classes `asyncio.Future` e `concurrent.futures.Future`, em sua maior parte, têm a mesma interface, mas são implementadas de forma diferente e não são intercambiáveis. A *PEP 3156 – Asynchronous IO Support Rebooted: the “asyncio” Module* (Suporte renovado a E/S assíncrona: o módulo “asyncio”, <https://www.python.org/dev/peps/pep-3156/>) tem o seguinte a dizer sobre essa situação infeliz:

No futuro (trocadilho intencional), talvez possamos unificar `asyncio.Future` e `concurrent.futures.Future` (por exemplo, acrescentando um método `__iter__` nesse último, que funcione com `yield from`).

Conforme mencionado na seção “Onde estão os futures?” na página 566, os futures são criados somente como resultado de escalonar algo para execução. Em `asyncio`, `BaseEventLoop.create_task(...)` recebe uma corrotina, escalona-a para execução e devolve uma instância de `asyncio.Task` – que é também uma instância de `asyncio.Future`, pois `Task` é uma subclasse de `Future` projetada para encapsular uma corrotina. Isso é semelhante a criar instâncias de `concurrent.futures.Future` chamando `Executor.submit(...)`.

Assim como sua contrapartida `concurrent.futures.Future`, a classe `asyncio.Future` oferece os métodos `.done()`, `.add_done_callback(...)` e `.results()`, entre outros. Os dois primeiros métodos funcionam conforme descritos em “Onde estão os futures” na página 566, mas `.result()` é bem diferente.

Em `asyncio.Future`, o método `.result()` não aceita argumentos, portanto você não pode especificar um `timeout`. Além disso, se chamar `.result()` e o future não tiver terminado, ele não ficará bloqueado à espera do resultado. Em vez disso, a exceção `asyncio.InvalidStateError` é levantada.

No entanto a maneira usual de obter o resultado de um `asyncio.Future` é com `yield from`, como veremos no exemplo 18.8.

Usar `yield from` com um future trata automaticamente de esperar pelo seu término, sem bloquear o loop de eventos – porque em `asyncio`, `yield from` é usado para devolver o controle ao loop de eventos.

Observe que usar `yield from` com um future é o equivalente à funcionalidade oferecida por `add_done_callback`, mas com corrotinas: em vez de disparar uma callback, quando a operação demorada termina, o loop de eventos define o resultado do future e a expressão `yield from` produz um valor de retorno em nossa corrotina suspensa, permitindo que ela retome a execução.

Em suma, como `asyncio.Future` foi projetado para funcionar com `yield from`, os seguintes métodos, com frequência, não são necessários:

- `my_future.add_done_callback(...)`, pois você pode simplesmente colocar qualquer processamento que seria feito após o término do future nas linhas depois de `yield from my_future` em sua corrotina. Essa é a grande vantagem das corrotinas: funções que possam ser suspensas e retomadas.
- `my_future.result()`, pois o valor de uma expressão `yield from` em um future é o resultado desse future (por exemplo, `result = yield from my_future`).

É claro que há situações em que `.done()`, `.add_done_callback(...)` e `.results()` são úteis. Mas, em uso normal, os futures de `asyncio` são acionados por `yield from`, e não pela chamada a esse métodos.

Vamos agora ver como `yield from` e a API de `asyncio` unificam futures, tasks e corrotinas.

yield from com futures, tasks e corrotinas

Em `asyncio`, há um relacionamento próximo entre futures e corrotinas, pois você pode obter o resultado de um `asyncio.Future` usando `yield from` nesse future. Isso quer dizer que `res = yield from foo()` funciona se `foo` é uma função de corrotina (sendo assim, ela devolve um objeto corrotina quando é chamada) ou se `foo` é uma função comum que devolve uma instância de `Future` ou de `Task`. Esse é um dos motivos pelos quais corrotinas e futures são intercambiáveis em muitas partes da API de `asyncio`.

Para executar, uma corrotina deve ser escalonada, e então ela é encapsulada em uma `asyncio.Task`. Dada uma corrotina, há dois modos principais de obter uma `Task`:

```
asyncio.async(coro_or_future, *, loop=None)4
```

Essa função unifica corrotinas e futures: o primeiro argumento pode ser um ou outro. Se for um `Future` ou uma `Task`, ele é devolvido sem alteração. Se for uma corrotina, `async` chama `loop.create_task(...)` nela para criar uma `Task`. Um loop de eventos opcional pode ser passado como o argumento nomeado `loop=`; se for omitido, `async` obterá o objeto loop chamando `asyncio.get_event_loop()`.

```
BaseEventLoop.create_task(coro)
```

Esse método escala a corrotina para execução e devolve um objeto `asyncio.Task`. Se for chamado em uma subclasse especializada de `BaseEventLoop`, o objeto devolvido poderá ser uma instância de outra classe compatível com `Task` oferecida por uma biblioteca externa (por exemplo, `Tornado`).

⁴ Nota do autor/revisor técnico: Como já observado antes, a partir do Python 3.4.4, deve-se usar `@asyncio.ensure_future` e não mais `@asyncio.async`. O novo nome é mais descritivo e não inclui o identificador `async`, que será uma palavra reservada futuramente.



`BaseEventLoop.create_task(...)` está disponível somente em Python 3.4.2 ou em versões mais recentes. Se você estiver usando uma versão mais antiga de Python 3.3 ou 3.4, utilize `asyncio.async(...)` ou instale uma versão mais recente de `asyncio` obtida no PyPI (<https://pypi.python.org/pypi/asyncio>).

Várias funções de `asyncio` aceitam corrotinas e as encapsulam em objetos `asyncio.Task` de forma automática usando `asyncio.async` internamente. Um exemplo é `BaseEventLoop.run_until_complete(...)`.

Se quiser fazer experimentos com futures e corrotinas no console de Python ou em testes rápidos, você pode usar o trecho de código a seguir:⁵

```
>>> import asyncio
>>> def run_sync(coro_or_future):
...     loop = asyncio.get_event_loop()
...     return loop.run_until_complete(coro_or_future)
...
>>> a = run_sync(some_coroutine())
```

A relação entre corrotinas, futures e tasks está documentada na seção 18.5.3. *Tasks and coroutines* (Tasks e corrotinas, <https://docs.python.org/3/library/asyncio-task.html>) da documentação de `asyncio`, na qual você verá a seguinte nota:

Nesta documentação, alguns métodos são referenciados como corrotinas, apesar de serem funções comuns devolvendo um `Future`. Isso é proposital para preservar a liberdade de ajustar a implementação dessas funções futuramente.

Após a discussão desses conceitos fundamentais, vamos agora estudar o código do script assíncrono de download de bandeiras, `flags_asyncio.py`, apresentado juntamente com os scripts sequencial e com pool de threads no exemplo 171 (Capítulo 17).

Fazendo download com `asyncio` e `aiohttp`

Até Python 3.4, `asyncio` suporta apenas TCP e UDP diretamente. Para HTTP ou qualquer outro protocolo, precisamos de pacotes de terceiros; `aiohttp` é o pacote que todos parecem estar usando para clientes e servidores HTTP com `asyncio` atualmente.

O exemplo 18.5 tem uma listagem completa do script `flags_asyncio.py` para download de bandeiras. Eis uma visão geral de como ele funciona:

1. Iniciamos o processo em `download_many` alimentando o loop de eventos com vários objetos corrotina produzidos por chamadas a `download_one`.

⁵ Sugerido por Petr Viktorin em uma mensagem de 11 de setembro de 2014 (<http://bit.ly/1JlwJmc>) na lista Python-ideas.

2. O loop de eventos de `asyncio` aciona uma corrotina de cada vez.
3. Quando uma corrotina cliente, como `get_flag`, usa `yield from` para acionar uma corrotina da biblioteca – por exemplo, `aihttp.request` –, o controle volta para o loop de eventos, que pode executar outra corrotina escalonada anteriormente.
4. O loop de eventos usa APIs de baixo nível baseadas em `callbacks` para ser notificado quando uma operação bloqueante é concluída.
5. Quando isso acontece, o laço principal envia um resultado à corrotina suspensa.
6. A corrotina então avança para o próximo `yield`, por exemplo, `yield from resp.read()` em `get_flag`. O loop de eventos assume o controle novamente. Os passos 4, 5 e 6 se repetem até o loop de eventos terminar.

Isso é semelhante ao exemplo que vimos na seção “A simulação da frota de táxis” na página 545, em que um laço principal iniciava vários processos associados a táxis, um de cada vez. À medida que cada processo de táxi cedia o controle, o laço principal escalonava o próximo evento para esse táxi (para ocorrer no futuro) e prosseguia acionando o próximo táxi na fila. A simulação de táxis é muito mais simples, e é mais fácil entender seu laço principal. Mas o fluxo geral é o mesmo em `asyncio`: um programa com uma só `thread` (`single-threaded`) em que um laço principal aciona corrotinas armazenadas em uma fila, uma por uma. Cada corrotina avança alguns passos e então devolve o controle ao laço principal, que, por sua vez, aciona a próxima corrotina da fila.

Vamos agora analisar o exemplo 18.5 passo a passo.

Exemplo 18.5 – `flags_asyncio.py`: script para download assíncrono com `asyncio` e `aihttp`

```
import asyncio
import aiohttp ❶

from flags import BASE_URL, save_flag, show, main ❷

@asyncio.coroutine ❸
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url) ❹
    image = yield from resp.read() ❺
    return image

@asyncio.coroutine
def download_one(cc): ❻
    image = yield from get_flag(cc) ❼
```

```

show(cc)
save_flag(image, cc.lower() + '.gif')
return cc

```

```

def download_many(cc_list):
    loop = asyncio.get_event_loop() ❸
    to_do = [download_one(cc) for cc in sorted(cc_list)] ❹
    wait_coro = asyncio.wait(to_do) ❺
    res, _ = loop.run_until_complete(wait_coro) ❻
    loop.close() ❼
    return len(res)

```

```

if __name__ == '__main__':
    main(download_many)

```

- ❶ aiohttp precisa ser instalado – não está na biblioteca-padrão.
- ❷ Reutiliza algumas funções do módulo `flags` (Exemplo 17.2).
- ❸ Corrotinas devem ser decoradas com `@asyncio.coroutine`.
- ❹ Operações bloqueantes são implementadas como corrotinas e seu código as aciona por meio de `yield from` para que executem assincronamente.
- ❺ Ler o conteúdo da resposta é uma operação assíncrona separada.
- ❻ `download_one` também precisa ser uma corrotina, pois usa `yield from`.
- ❼ A única diferença em relação à implementação sequencial de `download_one` são as palavras `yield from` nessa linha; o restante do corpo da função é *exatamente* o mesmo.
- ❽ Obtém uma referência à implementação do loop de eventos subjacente.
- ❾ Cria uma lista de objetos geradores/corrotinas chamando a função `download_one` uma vez para cada bandeira a ser obtida.
- ❺ Apesar do nome, `wait` não é uma função bloqueante. É uma corrotina que termina quando todas as corrotinas passadas para ela terminam (esse é o comportamento-padrão de `wait`; veja a explicação após esse exemplo).
- ❻ Executa o loop de eventos até `wait_coro` terminar, é aqui que o script fica bloqueado enquanto o loop de eventos executa. Ignoramos o segundo item devolvido por `run_until_complete`. O motivo será explicado a seguir.
- ❼ Encerra o loop de eventos.



Seria bom se instâncias de loops de evento fossem gerenciadores de contexto, de modo que pudéssemos usar um bloco `with` para garantir que o laço fosse encerrado. Contudo a situação se complica porque o código do cliente jamais cria o loop de eventos diretamente, mas obtém uma referência a ele chamando `asyncio.get_event_loop()`. Às vezes, nosso código não é “dono” do loop de eventos, portanto seria errado encerrá-lo. Por exemplo, ao usar um loop de eventos externo de uma GUI com um pacote como Quamash (<https://pypi.python.org/pypi/Quamash/>), a biblioteca Qt é responsável por encerrar o loop de eventos quando a aplicação termina.

A corrotina `asyncio.wait(...)` aceita um iterável de futures ou corrotinas; `wait` encapsula cada corrotina em uma `Task`. O resultado final é que todos os objetos gerenciados por `wait` tornam-se instâncias de `Future`, de uma maneira ou de outra. Por ser uma função de corrotina, chamar `wait(...)` devolve um objeto gerador/corrotina; este é o valor da variável `wait_coro`. Para acionar essa corrotina, nós a passamos para `loop.run_until_complete(...)`.

A função `loop.run_until_complete` aceita um future ou uma corrotina. Se receber uma corrotina, `run_until_complete` a encapsula em uma `Task`, de modo semelhante ao que `wait` faz. Corrotinas, futures e tasks podem ser todos acionados por `yield from`, e é isso que `run_until_complete` faz com o objeto `wait_coro` devolvido pela chamada a `wait`. Quando `wait_coro` executa até o final, `run_until_complete` devolve uma tupla de dois elementos em que o primeiro item é o conjunto dos futures concluídos e o segundo é o conjunto dos futures não concluídos. No exemplo 18.5, o segundo conjunto sempre estará vazio – é por isso que nós o ignoramos explicitamente atribuindo-o a `_`. Porém `wait` aceita dois argumentos exclusivamente nomeados que podem fazê-la retornar mesmo se houver algum future não concluído: `timeout` e `return_when`. Consulte a documentação de `asyncio.wait` (<http://bit.ly/1JlwZS2>) para ver os detalhes.

Observe que, no exemplo 18.5, não pude reutilizar a função `get_flag` de `flags.py` (Exemplo 17.2) porque ela usa a biblioteca `requests`, que faz E/S bloqueante. Para aproveitar `asyncio`, precisamos substituir toda função que acesse a rede por uma versão assíncrona acionada com `yield from` para que o controle seja devolvido ao loop de eventos. Usar `yield from` em `get_flag` significa que ela deve ser acionada como uma corrotina.

Pelo mesmo motivo, não pude reutilizar a função `download_one` de `flags_threadpool.py` (Exemplo 17.3) também. O código do exemplo 18.5 aciona `get_flag` com `yield from`, portanto `download_one` é, ela própria, uma corrotina. Para cada requisição, um objeto corrotina `download_one` é criado em `download_many`, e todos são acionados pela função `loop.run_until_complete`, depois de terem sido encapsulados pela corrotina `asyncio.wait`.

Há vários conceitos novos para compreender em `asyncio`, mas a lógica geral do exemplo 18.5 é fácil de seguir se você empregar um truque sugerido pelo próprio Guido van

Rossum: feche um olho e finja que as palavras reservadas `yield from` não estão lá. Se fizer isso, perceberá que o código é tão fácil de ler quanto um bom e velho código sequencial.

Por exemplo, imagine que o corpo desta corrotina...

```
@asyncio.coroutine
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    image = yield from resp.read()
    return image
```

... funciona como a função a seguir, exceto por jamais bloquear:

```
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = aiohttp.request('GET', url)
    image = resp.read()
    return image
```

Usar a sintaxe `yield from foo` evita bloqueios porque a corrotina atual é suspensa (isto é, o gerador delegante, no ponto em que está o código `yield from`), mas o fluxo de controle retorna ao loop de eventos, que pode acionar outras corrotinas. Quando o `future` ou a corrotina `foo` terminar, ele ou ela devolverá um resultado à corrotina suspensa, fazendo-a retomar a execução.

No final da seção “Usando `yield from`” na página 530, apresentei dois fatos válidos para todo uso de `yield from`. Veja-os resumidamente:

- Qualquer construção de corrotinas encadeadas com `yield from`, em última instância, precisa ser acionada por um chamador (`caller`) que não é uma corrotina; ele chama `next(...)` ou `.send(...)` no gerador mais externo que faz a delegação, de forma explícita ou implícita (por exemplo, em um laço `for`).
- O subgerador mais interno da cadeia deve ser um gerador simples, que usa somente `yield` – ou um objeto iterável.

Ao usar `yield from` com a API de `asyncio`, ambos os fatos permanecem válidos, com as seguintes especificidades:

- As cadeias de corrotinas que escrevemos são sempre acionadas passando nosso gerador mais externo, responsável pela delegação, a uma chamada de API de `asyncio`, por exemplo, `loop.run_until_complete(...)`. Em outras palavras, quando usamos `asyncio`, nosso código não aciona uma cadeia de corrotinas chamando `next(...)` ou `.send(...)` – o loop de eventos de `asyncio` faz isso.

- As cadeias de corrotinas que escrevemos sempre terminam delegando com `yield from` a alguma função ou um método de corrotina de `asyncio` (por exemplo, `yield from asyncio.sleep(...)` no exemplo 18.2) ou corrotinas de bibliotecas que implementam protocolos de níveis mais altos (por exemplo, `resp = yield from aiohttp.request('GET', url)` na corrotina `get_flag` do exemplo 18.5).

Em outras palavras, o subgerador mais interno será uma função de biblioteca que executa a E/S propriamente dita, e não um código que escrevemos.

Para resumir: quando usamos `asyncio`, nosso código assíncrono é constituído de corrotinas que são geradores delegantes, acionados pela própria `asyncio` e que, em última instância, delegam para corrotinas da biblioteca `asyncio` – provavelmente por meio de alguma biblioteca de terceiros como `aiohttp`. Essa organização cria pipelines em que o loop de eventos de `asyncio` aciona – por meio de nossas corrotinas – as funções de biblioteca que executam E/S assíncrona de baixo nível.

Agora estamos prontos para responder a uma pergunta feita no capítulo 17:

- Como `flags_asyncio.py` é capaz de executar cinco vezes mais rápido que `flags.py` se ambos têm uma só thread?

Dando voltas em chamadas bloqueantes

Ryan Dahl, criador do Node.js, apresenta a filosofia de seu projeto dizendo: “Estamos fazendo E/S de forma totalmente errada.”⁶ Ele define uma *função bloqueante* como aquela que faz E/S em disco ou em rede e argumenta que não podemos tratá-las como tratamos funções não bloqueantes. Para explicar por quê, ele apresenta os números das duas primeiras colunas da tabela 18.1.

Tabela 18.1 – Latência em computadores modernos para ler dados de diferentes dispositivos; a terceira coluna mostra os tempos proporcionais em uma escala mais fácil de entender para nós, seres humanos lentos.

Dispositivo	Ciclos de CPU	Escala proporcional para “seres humanos”
cache L1	3	3 segundos
cache L2	14	14 segundos
RAM	250	250 segundos
disco	41.000.000	1,3 ano
rede	240.000.000	7,6 anos

6 Video: *Introduction to Node.js* (Introdução ao Node.js, <https://www.youtube.com/watch?v=M-sc73Y-zQA>) em 4min55s.

Para entender a tabela 18.1, tenha em mente que CPUs modernas com clocks de GHz executam bilhões de ciclos por segundo. Vamos supor que uma CPU execute exatamente um bilhão de ciclos por segundo. Essa CPU pode executar 333.333.333 de leituras em cache L1 em um segundo, ou 4 (quatro!) leituras de rede no mesmo tempo. A terceira coluna da tabela 18.1 coloca esses números em perspectiva multiplicando a segunda coluna por um fator constante. Assim, em um universo alternativo, se uma leitura de cache L1 demorar 3 segundos, uma leitura de rede demoraria 76 anos!

Há duas maneiras de impedir que chamadas bloqueantes interrompam o progresso da aplicação como um todo:

- Executar cada operação bloqueante em uma thread separada.
- Transformar toda operação bloqueante em uma chamada assíncrona não bloqueante.

Threads funcionam bem, mas o overhead de memória para cada thread do sistema operacional – o tipo usado por Python – é da ordem de megabytes, dependendo do sistema operacional. Não podemos nos dar o luxo de ter uma thread por conexão se estivermos tratando milhares de conexões.

As callbacks são a maneira tradicional de implementar chamadas assíncronas com pouco overhead de memória. Elas são um conceito de baixo nível, semelhante ao mais antigo e mais primitivo mecanismo de concorrência: as interrupções de hardware. Em vez de esperar uma resposta, registramos uma função a ser chamada quando algo acontecer. Desse modo, toda chamada que fizermos pode ser não bloqueante. Ryan Dahl defende as callbacks pela sua simplicidade e pelo baixo overhead.

É claro que só podemos fazer as callbacks funcionarem porque o loop de eventos subjacente às nossas aplicações assíncronas pode contar com uma infraestrutura que usa interrupções, threads, polling, processos em background, etc., para garantir que várias requisições concorrentes progridam e, em algum momento, sejam concluídas.⁷ Quando o loop de eventos obtém uma resposta, ele chama nosso código novamente. Porém a thread principal única compartilhada pelo loop de eventos e o código de nossa aplicação jamais ficarão bloqueados – se não cometermos erros.

Quando usados como corrotinas, os geradores oferecem uma maneira alternativa de fazer programação assíncrona. Do ponto de vista do loop de eventos, chamar uma callback ou chamar `.send()` em uma corrotina suspensa é praticamente o mesmo. Há um overhead de memória para cada corrotina suspensa, mas é várias ordens de grandeza menor que o overhead de cada thread. Além disso, as corrotinas evitam o

⁷ De fato, embora Node.js não suporte threads de nível de usuário escritas em JavaScript, internamente, ele implementa um pool de threads em C com a biblioteca libuv para oferecer suas APIs de arquivo baseadas em callback – porque atualmente (em 2014) não há APIs assíncronas estáveis e portáveis para tratamento de arquivos na maioria dos sistemas operacionais.

temido “inferno de callbacks” (callback hell), que discutiremos na seção “De callbacks a futures e corrotinas” na página 620.

Agora o ganho de cinco vezes no desempenho de *flags_asyncio.py* sobre *flags.py* deve fazer sentido: *flags.py* gasta bilhões de ciclos de CPU esperando cada download, um após o outro. A CPU, na verdade, faz muitas tarefas nesse meio-tempo; ela só não executa o seu programa. Em comparação, quando *loop_until_complete* é chamado na função *download_many* de *flags_asyncio.py*, o loop de eventos conduz cada corrotina *download_one* até o primeiro *yield from* que, por sua vez, conduz cada corrotina *get_flag* até o primeiro *yield from*, chamando *aihttp.request(...)*. Nenhuma dessas chamadas é bloqueante, portanto todas as requisições são iniciadas em uma fração de segundo.

Quando a infraestrutura de *asyncio* obtém a primeira resposta de volta, o loop de eventos a envia para uma corrotina *get_flag* em espera. Quando *get_flag* obtém uma resposta, ela avança para o próximo *yield from*, que chama *resp.read()* e devolve o controle ao laço principal. Outras respostas chegam em rápida sucessão (porque foram feitas quase no mesmo instante). À medida que cada *get_flag* retorna, o gerador *download_one* que fez a delegação retoma a execução e salva o arquivo de imagem.



Para ter o máximo de desempenho, a operação *save_flag* deveria ser assíncrona, mas atualmente *asyncio* não oferece uma API assíncrona para lidar com arquivos – como faz o Node. Se isso se tornar um gargalo em sua aplicação, você pode usar a função *loop.run_in_executor* (<http://bit.ly/1HGtQzc>) para executar *save_flag* em um pool de threads. O exemplo 18.9 mostra como isso é feito.

Como as operações assíncronas são intercaladas, o tempo total necessário para o download de várias imagens de forma concorrente é muito menor que fazê-lo sequencialmente. Ao fazer 600 requisições HTTP com *asyncio*, recebi todos os resultados mais de 70 vezes mais rápido em comparação com um script sequencial.

Vamos agora voltar para o exemplo do cliente HTTP para ver como podemos exibir uma barra de progresso animada e realizar um tratamento de erros adequado.

Melhorando o script para download com *asyncio*

Lembre-se de que, de acordo com a seção “Downloads com exibição de progresso e tratamento de erros” na página 576, a série de exemplos *flags2* compartilha a mesma interface de linha de comando. Isso inclui *flags2_asyncio.py*, que analisaremos nesta seção. O exemplo 18.6 mostra como obter 100 bandeiras (*-a1 100*) do servidor ERROR usando 100 requisições concorrentes (*-m 100*).

Exemplo 18.6 – Executando `flags2_asyncio.py`

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
.....
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

**Aja responsabilmente quando testar clientes concorrentes**

Mesmo que o tempo total de download não seja diferente entre clientes com threads e com `asyncio`, `asyncio` pode enviar requisições mais rapidamente, portanto é mais provável ainda que o servidor suspeite de um ataque DoS. Para realmente exercitar esses clientes concorrentes com velocidade máxima, configure um servidor HTTP local para testes, conforme explicado em *README.rst* (<http://bit.ly/1Jlsg2L>) no diretório *17-futures/countries/* (<http://bit.ly/1f6ChKk>) do repositório de código de *Python fluente* (<http://bit.ly/1JltSu>).

Agora vamos ver como `flags2_asyncio.py` está implementado.

Usando `asyncio.as_completed`

No exemplo 18.5, passei uma lista de corrotinas para `asyncio.wait`, que – quando acionada por `loop.run_until_complete` – retornava os resultados dos downloads quando todos estavam concluídos. Contudo, para atualizar uma barra de progresso, precisamos obter os resultados à medida que estiverem prontos. Felizmente, existe um equivalente em `asyncio` para a função geradora `as_completed` usada no exemplo com pool de threads com a barra de progresso (Exemplo 17.14).

Escrever um exemplo da série `flags2` para aproveitar `asyncio` implica reescrever diversas funções que a versão com `concurrent.future` conseguiu reutilizar. Isso ocorre porque há apenas uma thread principal em um programa com `asyncio`, e não podemos nos dar o luxo de ter chamadas bloqueantes nessa thread, pois é a mesma thread que executa o loop de eventos. Então tive de reescrever `get_flag` para que usasse `yield from` para todo acesso à rede. Agora `get_flag` é uma corrotina, portanto `download_one` deve acioná-la com `yield from`; desse modo, `download_one` também se transforma em uma corrotina. Antes, no exemplo 18.5, `download_one` era acionada por `download_many`: as chamadas a `download_one` eram encapsuladas em uma chamada a `asyncio.wait` e passadas para `loop.run_until_complete`. Agora precisamos ter um controle mais preciso para informar o progresso e tratar erros, portanto transferi a maior parte da lógica de `download_many`

para uma nova corrotina `download_coro` e uso `download_many` somente para criar o loop de eventos e escalonar `download_coro`.

O exemplo 18.7 mostra o início do script *flags2_asyncio.py*, em que as corrotinas `get_flag` e `download_one` estão definidas. O exemplo 18.8 lista o restante do código-fonte, com `download_coro` e `download_many`.

Exemplo 18.7 – *flags2_asyncio.py*: parte inicial do script; o código restante está no exemplo 18.8

```
import asyncio
import collections

import aiohttp
from aiohttp import web
import tqdm
from flags2_common import main, HTTPStatus, Result, save_flag

# default definido com um valor baixo para evitar erros do site remoto, por exemplo
# 503 - Service Temporarily Unavailable (Serviço temporariamente indisponível)
DEFAULT_CONCUR_REQ = 5
MAX_CONCUR_REQ = 1000

class FetchError(Exception): ❶
    def __init__(self, country_code):
        self.country_code = country_code

@asyncio.coroutine
def get_flag(base_url, cc): ❷
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    if resp.status == 200:
        image = yield from resp.read()
        return image
    elif resp.status == 404:
        raise web.HTTPNotFound()
    else:
        raise aiohttp.HttpProcessingError(
            code=resp.status, message=resp.reason,
            headers=resp.headers)

@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose): ❸
    try:
        with (yield from semaphore): ❹
```



```

        image = yield from get_flag(base_url, cc) ❸
    except web.HTTPNotFound: ❹
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc ❺
    else:
        save_flag(image, cc.lower() + '.gif') ❻
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)

```

- ❶ Essa exceção especial será usada para encapsular outra exceção HTTP ou de rede e armazenar o valor de `country_code` para reportar erros.
- ❷ `get_flag` devolve os bytes da imagem baixada, levanta `web.HTTPNotFound` se o status da resposta HTTP for 404 ou levanta `aihttp.HttpProcessingError` para outros códigos de status HTTP.
- ❸ O argumento `semaphore` é uma instância de `asyncio.Semaphore` (<http://bit.ly/1f6Csp8>) – um dispositivo de sincronização que limita o número de requisições concorrentes.
- ❹ Um `semaphore` é usado como gerenciador de contexto em uma expressão `yield from` para que o sistema como um todo não fique bloqueado: somente essa corrotina será bloqueada enquanto o contador do semáforo estiver com o número máximo permitido.
- ❺ Quando esse comando `with` sai, o contador de `semaphore` é decrementado, desbloqueando outra instância de corrotina que possa estar esperando o mesmo objeto `semaphore`.
- ❻ Se a bandeira não tiver sido encontrada, simplesmente define o status de `Result` de acordo com isso.
- ❼ Qualquer outra exceção será informada como um `FetchError`, com o código do país e a exceção original encadeada usando a sintaxe `raise X from Y` introduzida na *PEP 3134 – Exception Chaining and Embedded Tracebacks* (Encadeamento de exceções e `tracebacks` embutidos, <https://www.python.org/dev/peps/pep-3134/>).
- ❽ Essa chamada de função salva a imagem da bandeira no disco.

No exemplo 18.7, podemos ver que o código de `get_flag` e de `download_one` mudou significativamente em relação à versão sequencial porque essas funções agora são corrotinas que usam `yield from` para fazer chamadas assíncronas.

O tipo de cliente de rede que estamos estudando sempre deve usar algum mecanismo de limitação (throttling) para evitar sobrecarregar o servidor com muitas requisições concorrentes – o desempenho geral do sistema pode se degradar se o servidor ficar sobrecarregado. Em *flags2_threadpool.py* (Exemplo 17.14), a limitação foi feita instanciando `ThreadPoolExecutor` com o argumento obrigatório `max_workers` definido como `concur_req` na função `download_many`, portanto somente `concur_req` threads são iniciadas no pool. Em *flags2_asyncio.py*, usei um `asyncio.Semaphore`, criado pela função `downloader_coro` (mostrada a seguir no exemplo 18.8) e passado como o argumento `semaphore` a `download_one` no exemplo 18.7.⁸

Um `Semaphore` é um objeto que armazena um contador interno que é decrementado sempre que chamamos o método/corrotina `.acquire()` nesse objeto, e é incrementado sempre que chamamos o método/corrotina `.release()`. O valor inicial do contador é definido quando `Semaphore` é instanciado, como nesta linha de `downloader_coro`:

```
semaphore = asyncio.Semaphore(concur_req)
```

Chamar `.acquire()` não causa um bloqueio quando o contador é maior que zero, mas se o contador for zero, `.acquire()` bloqueará a corrotina que fizer a chamada até que outra corrotina chame `.release()` no mesmo `Semaphore`, decrementando assim o contador. No exemplo 18.7, não chamo `.acquire()` nem `.release()`, mas uso `semaphore` como um gerenciador de contexto neste bloco de código em `download_one`:

```
with (yield from semaphore):  
    image = yield from get_flag(base_url, cc)
```

Esse trecho de código garante que não mais que `concur_req` instâncias de corrotinas `get_flags` sejam iniciadas ao mesmo tempo.

Vamos agora dar uma olhada no restante do script no exemplo 18.8. Observe que a maior parte das operações da função `download_many` anterior agora está em uma corrotina, `downloader_coro`. Isso foi necessário porque precisamos usar `yield from` para recuperar os resultados dos futures produzidos por `asyncio.as_completed`; desse modo, `as_completed` deve ser chamado em uma corrotina. Entretanto eu não podia simplesmente transformar `download_many` em uma corrotina porque preciso passá-la para a função `main` de *flags2_common* na última linha do script, e essa função `main` não espera uma corrotina, mas uma função comum. Sendo assim, criei `downloader_coro` para executar o laço `as_completed`; agora `download_many` simplesmente cria o loop de eventos e escalona `downloader_coro` passando-a para `loop.run_until_complete`.

⁸ Agradeço a Guto Maia que percebeu que `Semaphore` não havia sido explicado na versão preliminar deste livro.

Exemplo 18.8 – flags2_asyncio.py: continuação do script do exemplo 18.7

```
@asyncio.coroutine
def downloader_coro(cc_list, base_url, verbose, concur_req): ❶
    counter = collections.Counter()
    semaphore = asyncio.Semaphore(concur_req) ❷
    to_do = [download_one(cc, base_url, semaphore, verbose)
              for cc in sorted(cc_list)] ❸

    to_do_iter = asyncio.as_completed(to_do) ❹
    if not verbose:
        to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list)) ❺
    for future in to_do_iter: ❻
        try:
            res = yield from future ❼
        except FetchError as exc: ❽
            country_code = exc.country_code ❾
            try:
                error_msg = exc.__cause__.args[0] ❿
            except IndexError:
                error_msg = exc.__cause__.__class__.__name__ ⓫
            if verbose and error_msg:
                msg = '*** Error for {}: {}'.format(country_code, error_msg)
                print(msg)
            status = HTTPStatus.error
        else:
            status = res.status

        counter[status] += 1 ⓬

    return counter ⓭

def download_many(cc_list, base_url, verbose, concur_req):
    loop = asyncio.get_event_loop()
    coro = downloader_coro(cc_list, base_url, verbose, concur_req)
    counts = loop.run_until_complete(coro) ⓮
    loop.close() ⓯

    return counts

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

- ❶ A corrotina recebe os mesmos argumentos que `download_many`, mas não pode ser chamada diretamente a partir de `main`, exatamente por ser uma função de corrotina, e não uma função comum como `download_many`.

- ❷ Cria um `asyncio.Semaphore` que permitirá até `concur_req` corrotinas ativas entre aquelas que usam esse semáforo.
- ❸ Cria uma lista de objetos corrotina, um para cada chamada à corrotina `download_one`.
- ❹ Obtém um iterador que devolverá futures à medida que terminarem.
- ❺ Encapsula o iterador na função `tqdm` para exibir o progresso.
- ❻ Itera pelos futures concluídos; esse laço é bem parecido com aquele em `download_many` no exemplo 17.14; a maioria das alterações tem a ver com tratamento de exceções por causa das diferenças nas bibliotecas HTTP (*requests versus aiohttp*).
- ❼ A maneira mais fácil de recuperar o resultado de um `asyncio.Future` é usar `yield from` em vez de chamar `future.result()`.
- ❽ Toda exceção em `download_one` é encapsulada em um `FetchError` com a exceção original encadeada.
- ❾ Obtém o código do país em que o erro ocorreu a partir da exceção `FetchError`.
- ❿ Tenta recuperar a mensagem de erro da exceção original (`__cause__`).
- ⓫ Se a mensagem de erro não puder ser encontrada na exceção original, usa o nome da classe da exceção encadeada como mensagem de erro.
- ⓬ Conta os resultados.
- ⓭ Devolve o contador, conforme feito nos outros scripts.
- ⓮ `download_many` simplesmente instancia a corrotina e passa-a para o loop de eventos com `run_until_complete`.
- ⓯ Quando todas as tarefas estiverem concluídas, encerrará o loop de eventos e devolverá `counts`.

No exemplo 18.8, não pudemos usar o mapeamento de futures para códigos de países que vimos no exemplo 17.14, pois os futures devolvidos por `asyncio.as_completed` não são necessariamente os mesmos que passamos para a chamada a `as_completed`. Internamente, a implementação de `asyncio` substitui os objetos futures que fornecemos por outros que, no final, produzirão os mesmos resultados.⁹

Como não pude usar os futures como chaves para recuperar o código do país de um `dict` em caso de falhas, implementei a exceção personalizada `FetchError` (mostrada no exemplo 18.7). `FetchError` encapsula uma exceção de rede e armazena o código do país

9 Uma discussão detalhada sobre esse assunto pode ser encontrada em uma lista de discussão que iniciei no grupo `python-tulip` cujo título é “Which other futures my come out of `asyncio.as_completed`?” (Que outros futures podem vir de `asyncio.as_completed`?, <http://bit.ly/1f6CBZx>). Guido respondeu e deu esclarecimentos sobre a implementação de `as_completed`, assim como da íntima relação entre futures e corrotinas em `asyncio`.

associado a ela, de modo que esse código pode ser informado com o erro em modo verboso. Se não houver erros, o código do país estará disponível como resultado da expressão `yield from future` no início do laço `for`.

Com isso, encerramos a discussão do exemplo com `asyncio`, funcionalmente equivalente a `flags2_threadpool.py` que vimos antes. A seguir, implementaremos melhorias em `flags2_asyncio.py` que nos permitirão uma melhor exploração de `asyncio`.

Quando discutimos o exemplo 18.7, mencionei que `save_flag` faz E/S em disco e que deveria ser executada assincronamente. A seção a seguir mostra como isso pode ser feito.

Usando um executor para evitar bloqueio do loop de eventos

Na comunidade Python, temos a tendência de ignorar o fato de que o acesso ao sistema de arquivos local é bloqueante, justificando que ele não sofre do alto nível de latência do acesso à rede (que também é perigosamente imprevisível). Em comparação, programadores que usam Node.js são constantemente lembrados de que todas as funções do sistema de arquivos são bloqueantes porque suas assinaturas exigem uma callback. Lembre-se de que, de acordo com a tabela 18.1, o bloqueio para E/S em disco desperdiça milhões de ciclos de CPU, e isso pode ter um impacto significativo no desempenho da aplicação.

No exemplo 18.7, a função bloqueante é `save_flag`. Na versão do script com threads (Exemplo 17.14), `save_flag` bloqueia a thread que executa a função `download_one`, mas essa é apenas uma entre várias threads de trabalho. Sem que possamos ver, a chamada de E/S bloqueante libera a GIL, permitindo que outra thread possa prosseguir. Contudo, em `flags2_asyncio.py`, `save_flag` bloqueia a thread única que nosso código compartilha com o loop de eventos de `asyncio`; desse modo, a aplicação como um todo fica congelada enquanto o arquivo é salvo. A solução para esse problema está no método `run_in_executor` do objeto loop de eventos.

Internamente, o loop de eventos de `asyncio` tem um executor de pool de threads, e você pode enviar invocáveis para serem executados por ele com `run_in_executor`. Para usar esse recurso em nosso exemplo, somente algumas linhas precisam ser alteradas na corrotina `download_one`, como mostra o exemplo 18.9.

Exemplo 18.9 – `flags2_asyncio_executor.py`: usando o executor default de pool de threads para executar `save_flag`

```
@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore):
```

```

        image = yield from get_flag(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        loop = asyncio.get_event_loop() ❶
        loop.run_in_executor(None, ❷
            save_flag, image, cc.lower() + '.gif') ❸
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)

```

- ❶ Obtém uma referência para o objeto loop de eventos.
- ❷ O primeiro argumento de `run_in_executor` é uma instância do executor; se for `None`, o executor default de pool de threads do loop de eventos será usado.
- ❸ Os argumentos restantes são o invocável e seus argumentos posicionais.



Quando testei o exemplo 18.9, não houve nenhuma mudança perceptível de desempenho ao usar `run_in_executor` para salvar os arquivos de imagem porque eles não são grandes (13 KB cada, em média). No entanto você verá um efeito diferente se editar a função `save_flag` em `flags2_common.py` para que salve dez vezes mais bytes em cada arquivo – simplesmente escrevendo `fp.write(img*10)` em vez de `fp.write(img)`. Com um tamanho médio de download de 130 KB, a vantagem de usar `run_in_executor` torna-se mais evidente. Se você estiver fazendo download de imagens com megapixels, o ganho de velocidade será significativo.

A vantagem de usar corrotinas em vez de callbacks torna-se evidente quando precisamos coordenar requisições assíncronas, e não apenas fazer requisições totalmente independentes. A próxima seção explica o problema e a solução.

De callbacks a futures e corrotinas

Programação orientada a eventos com corrotinas exige um pouco de esforço para dominar, portanto é bom perceber claramente como ela representa melhorias em relação ao estilo clássico com callbacks. Esse é o tema desta seção.

Qualquer pessoa com alguma experiência em programação orientada a eventos usando callbacks conhece o termo “inferno de callbacks” (callback hell): ter callbacks aninhadas, em que uma operação depende do resultado da operação anterior. Se houver três chamadas assíncronas que devam ocorrer em sucessão, você precisará implementar callbacks aninhadas com três níveis de profundidade. O exemplo 18.10 mostra um exemplo em JavaScript.

Exemplo 18.10 – Inferno de callbacks em JavaScript: funções anônimas aninhadas, também conhecidas como Pyramid of Doom (Pirâmide da Perdição)

```
api_call1(request1, function (response1) {  
    // estágio 1  
    var request2 = step1(response1);  
    api_call2(request2, function (response2) {  
        // estágio 2  
        var request3 = step2(response2);  
        api_call3(request3, function (response3) {  
            // estágio 3  
            step3(response3);  
        });  
    });  
});
```

No exemplo 18.10, `api_call1`, `api_call2` e `api_call3` são funções de biblioteca que seu código usa para obter resultados assincronamente – talvez `api_call1` acesse um banco de dados e `api_call2` obtenha dados de um web service, por exemplo. Cada uma dessas funções aceita uma função de callback que, em JavaScript, muitas vezes, são funções anônimas (são chamadas de `stage1`, `stage2` e `stage3` no exemplo em Python a seguir). `step1`, `step2` e `step3`, nesse caso, representam funções comuns de sua aplicação, que processam as respostas recebidas pelas callbacks.

O exemplo 18.11 mostra como é o inferno de callbacks em Python.

Exemplo 18.11 – Inferno de callbacks em Python: callbacks encadeadas

```
def stage1(response1):  
    request2 = step1(response1)  
    api_call2(request2, stage2)  
  
def stage2(response2):  
    request3 = step2(response2)  
    api_call3(request3, stage3)
```



```
def stage3(response3):
    step3(response3)
```

```
api_call1(request1, stage1)
```

Embora o código do exemplo 18.11 esteja organizado de modo bem diferente do exemplo 18.10, eles fazem exatamente o mesmo, e o exemplo em JavaScript poderia ser escrito usando a mesma organização (mas o código em Python não pode ser escrito no estilo de JavaScript por causa das limitações sintáticas de `lambda`).

Códigos organizados como nos exemplos 18.10 ou 18.11 são difíceis de ler, mas são mais difíceis ainda de escrever: cada função faz parte do trabalho, define a próxima callback e retorna para deixar o loop de eventos prosseguir. A essa altura, todo o contexto local foi perdido. Quando a próxima callback (por exemplo, `stage2`) for executada, você não terá mais o valor de `request2`. Se precisar dele, você deverá contar com closures ou com estruturas de dados externas para armazená-lo entre os diferentes estágios do processamento.

Nesse contexto, as corrotinas ajudam muito. Em uma corrotina, para executar três ações assíncronas em sucessão, você usa `yield from` três vezes para permitir que o loop de eventos continue executando. Quando um resultado está pronto, a corrotina é acionada pelo loop de eventos com uma chamada a `.send()`. Do ponto de vista do loop de eventos, isso é semelhante a chamar uma callback. Entretanto, para os usuários de uma API assíncrona que use corrotinas, a situação é bem melhor: a sequência completa com as três operações fica em um só corpo de função, como um bom e velho código sequencial, com variáveis locais para preservar o contexto de toda a tarefa em andamento. Veja o exemplo 18.12.

Exemplo 18.12 – Corrotinas e `yield from` permitem programação assíncrona sem callbacks

```
@asyncio.coroutine
```

```
def three_stages(request1):
    response1 = yield from api_call1(request1)
    # estágio 1
    request2 = step1(response1)
    response2 = yield from api_call2(request2)
    # estágio 2
    request3 = step2(response2)
    response3 = yield from api_call3(request3)
    # estágio 3
    step3(response3)
```

```
loop.create_task(three_stages(request1)) # necessário escalonar explicitamente a execução
```

O exemplo 18.12 é muito mais fácil de seguir que os exemplos anteriores em JavaScript e Python: os três estágios da operação aparecem um após o outro, dentro da mesma função. Isso faz com que o uso de resultados anteriores seja trivial em processamentos que vêm depois. Também oferece um contexto para tratar erros por meio de exceções.

Suponha que no exemplo 18.11 o processamento da chamada a `api_call2(request2, stage2)` levante uma exceção de E/S (é a última linha da função `stage1`). A exceção não pode ser capturada em `stage1` porque `api_call2` é uma chamada assíncrona: ela retorna imediatamente, antes de qualquer E/S ser executada. Em APIs baseadas em `callback`, isso é resolvido registrando duas `callbacks` para cada chamada assíncrona: uma para tratar o resultado de operações bem-sucedidas e outra para tratar erros. As condições de trabalho no inferno de `callbacks` se deterioram rapidamente quando é preciso tratar de erros.

Em comparação, no exemplo 18.12, todas as chamadas assíncronas nessa operação de três estágios estão na mesma função, `three_stages`; se as chamadas assíncronas `api_call1`, `api_call2` e `api_call3` levantarem exceções, poderemos tratá-las colocando as respectivas linhas `yield from` em blocos `try/except`.

É muito melhor que o inferno de `callbacks`, mas não chamaria isso de paraíso das corrotinas, pois há um preço a ser pago. Em vez de funções comuns, você precisa usar corrotinas e acostumar-se com `yield from`, portanto esse é o primeiro obstáculo. Após escrever `yield from` em uma função, ela agora se torna uma corrotina e você não pode simplesmente chamá-la, como chamávamos `api_call1(request1, stage1)` no exemplo 18.11 para iniciar a cadeia de `callbacks`. Você deve explicitamente escalonar a execução da corrotina com o loop de eventos ou acioná-la usando `yield from` em outra corrotina escalonada para execução. Sem a chamada a `loop.create_task(three_stages(request1))` na última linha, nada aconteceria no exemplo 18.12.

O próximo exemplo coloca essa teoria em prática.

Fazendo várias requisições para cada download

Suponha que você queira salvar a bandeira de cada país com o nome e o código desse país, em vez de usar apenas o código do país. Você precisará fazer duas requisições HTTP por bandeira: uma para obter a imagem dela e outra para o arquivo *metadata.json* no mesmo diretório em que está a imagem: é aí que o nome do país está registrado.

Fazer a articulação entre várias requisições na mesma tarefa é fácil no script com `threads`: basta fazer uma requisição e depois a outra, bloqueando a `thread` duas vezes e mantendo ambos os dados (código do país e o nome) em variáveis locais, prontas para serem usadas quando os arquivos forem salvos. Se precisar fazer o mesmo em um script assíncrono com `callbacks`, você começará a sentir o cheiro de enxofre do

inferno de callbacks: o código e o nome do país deverão ser passados para uma closure ou mantidos em algum lugar até você poder salvar o arquivo, pois cada callback executa em um contexto local diferente. Corrotinas e `yield from` ajudam a resolver esse problema. A solução não é tão simples como no caso das threads, porém é mais legível que callbacks encadeadas ou aninhadas.

O exemplo 18.13 mostra o código da terceira variante do script de download de bandeiras com `asyncio`, que usa o nome do país para salvar cada bandeira. `download_many` e `downloader_coro` são os mesmos de *flags2_asyncio.py* (Exemplos 18.7 e 18.8). As mudanças estão em:

`download_one`

Essa corrotina agora usa `yield from` para delegar para `get_flag` e para a nova corrotina `get_country`.

`get_flag`

A maior parte do código dessa corrotina foi transferida para uma nova corrotina `http_get` para que possa também ser usada por `get_country`.

`get_country`

Essa corrotina busca o arquivo *metadata.json* do código do país e obtém daí o nome do país.

`http_get`

Código comum para obter um arquivo da Web.

Exemplo 18.13 – `flags3_asyncio.py`: mais delegação de corrotina para fazer duas requisições por bandeira

```
@asyncio.coroutine
```

```
def http_get(url):
```

```
    res = yield from aiohttp.request('GET', url)
```

```
    if res.status == 200:
```

```
        ctype = res.headers.get('Content-type', '').lower()
```

```
        if 'json' in ctype or url.endswith('json'):
```

```
            data = yield from res.json() ❶
```

```
        else:
```

```
            data = yield from res.read() ❷
```

```
        return data
```

```
    elif res.status == 404:
```

```
        raise web.HTTPNotFound()
```

```
    else:
```

```

        raise aiohttp.errors.HttpProcessingError(
            code=res.status, message=res.reason,
            headers=res.headers)

```

```
@asyncio.coroutine
```

```

def get_country(base_url, cc):
    url = '{}/{cc}/metadata.json'.format(base_url, cc=cc.lower())
    metadata = yield from http_get(url) ❸
    return metadata['country']

```

```
@asyncio.coroutine
```

```

def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    return (yield from http_get(url)) ❹

```

```
@asyncio.coroutine
```

```

def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore): ❺
            image = yield from get_flag(base_url, cc)
        with (yield from semaphore):
            country = yield from get_country(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        country = country.replace(' ', '_')
        filename = '{}-{}.gif'.format(country, cc)
        loop = asyncio.get_event_loop()
        loop.run_in_executor(None, save_flag, image, filename)
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)

```

- ❶ Se o tipo de conteúdo contém 'json' ou a url termina com .json, usa o método .json() da resposta para fazer parse dela e devolve uma estrutura de dados Python – nesse caso, um dict.

- ❷ Caso contrário, usa `.read()` para buscar os bytes como estão.
- ❸ `metadata` receberá um dict Python criado a partir do conteúdo JSON.
- ❹ Os parênteses externos nesse caso são necessários porque o parser de Python se confunde e produz um erro de sintaxe quando vê as palavras reservadas `return` `yield from` assim, em sequência.
- ❺ Coloquei as chamadas a `get_flag` e `get_country` em blocos `with` separados, controlados por `semaphore` porque queria retê-lo pelo mínimo de tempo possível.

A sintaxe `yield from` aparece nove vezes no exemplo 18.13. A essa altura, você já deve estar pegando o jeito de como essa construção é usada para delegar de uma corrotina para outra sem bloquear o loop de eventos.

O desafio é saber quando você deve usar `yield from` e quando não pode usá-lo. A resposta, a princípio, é simples: use `yield from` com corrotinas e instâncias de `asyncio.Future` – incluindo `tasks`. Porém algumas APIs são complicadas, misturando corrotinas e funções comuns de forma aparentemente arbitrária, como a classe `StreamWriter` que usaremos em um dos servidores na próxima seção.

O exemplo 18.13 encerra a série de exemplos `flags2`. Incentivo você a brincar com eles para desenvolver uma intuição de como clientes HTTP concorrentes funcionam. Use as opções de linha de comando `-a`, `-e` e `-l` para controlar o número de downloads, e a opção `-m` para definir o número de downloads concorrentes. Execute testes nos servidores `LOCAL`, `REMOTE`, `DELAY` e `ERROR`. Descubra o número ideal de downloads concorrentes para maximizar a taxa de transferência (throughput) em cada servidor. Ajuste as configurações do script `vaurien_error_delay.sh` para acrescentar ou remover erros e atrasos.

Vamos agora passar dos scripts clientes para a implementação de servidores com `asyncio`.

Escrevendo servidores com `asyncio`

O exemplo simples clássico de um servidor TCP é um servidor de eco. Implementaremos exemplos simples um pouco mais interessantes: buscadores de caracteres Unicode, inicialmente usando TCP puro e depois usando HTTP. Esses servidores permitirão aos clientes consultar caracteres Unicode com base em palavras presentes em seus nomes canônicos, usando o módulo `unicodedata` que discutimos na seção “Base de dados Unicode” na página 161. Uma sessão Telnet com o servidor TCP para localização de caracteres, em que peças de xadrez e caracteres com a palavra “sun” são procurados, está na figura 18.2.

```
lontra:charfinder luciano$ telnet localhost 2323
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^J'.
?> chess black
U+265A ♠      BLACK CHESS KING
U+265B ♚      BLACK CHESS QUEEN
U+265C ♜      BLACK CHESS ROOK
U+265D ♝      BLACK CHESS BISHOP
U+265E ♞      BLACK CHESS KNIGHT
U+265F ♟      BLACK CHESS PAWN
6 matches for 'chess black'
?> sun
U+2600 ☀      BLACK SUN WITH RAYS
U+2600 ☀      SUN
U+263C ☼      WHITE SUN WITH RAYS
U+26C5 ☂      SUN BEHIND CLOUD
U+2E9C ☞      CJK RADICAL SUN
U+2F47 ☞      KANGXI RADICAL SUN
U+3230 ☞      PARENTHEZIZED IDEOGRAPH SUN
U+3290 ☞      CIRCLED IDEOGRAPH SUN
U+C21C ☞      HANGUL SYLLABLE SUN
U+1F31E ☀     SUN WITH FACE
10 matches for 'sun'
?> ^C
Connection closed by foreign host.
lontra:charfinder luciano$
```

Figura 18.2 – Uma sessão Telnet com o servidor `tcp_charfinder.py`: consultando “chess black” e “sun”.

Agora vamos às implementações.

Um servidor TCP com asyncio

A maior parte da lógica desses exemplos está no módulo `charfinder.py`, que não tem nada de concorrente. Você pode usar `charfinder.py` como um buscador de caracteres utilizável pela linha de comando, mas, acima de tudo, ele foi projetado para fornecer conteúdo aos nossos servidores com `asyncio`. O código de `charfinder.py` está no repositório de código de *Python fluente* (<https://github.com/fluentpython/example-code>).

O módulo `charfinder` indexa cada palavra que aparece nos nomes dos caracteres da base de dados Unicode que vem com Python e cria um índice invertido armazenado em um dict. Por exemplo, a entrada do índice invertido para a chave 'SUN' contém um set com os dez caracteres Unicode que têm essa palavra em seus nomes. O índice invertido é salvo em um arquivo local `charfinder_index.pickle`. Se várias palavras aparecerem na consulta, `charfinder` calculará a intersecção dos conjuntos recuperados a partir do índice.

Vamos agora nos concentrar no script *tcp_charfinder.py* que responde às consultas da figura 18.2. Como tenho muito a dizer sobre esse código, separei-o em duas partes: o exemplo 18.14 e o exemplo 18.15.

Exemplo 18.14 – *tcp_charfinder.py*: um servidor TCP simples que usa `asyncio.start_server`; o código desse módulo continua no exemplo 18.15.

```
import sys
import asyncio

from charfinder import UnicodeNameIndex ❶

CRLF = b'\r\n'
PROMPT = b'?> '

index = UnicodeNameIndex() ❷

@asyncio.coroutine
def handle_queries(reader, writer): ❸
    while True: ❹
        writer.write(PROMPT) # não pode usar yield from! ❺
        yield from writer.drain() # precisa usar yield from! ❻
        data = yield from reader.readline() ❼
        try:
            query = data.decode().strip()
        except UnicodeDecodeError: ❽
            query = '\x00'
        client = writer.get_extra_info('peername') ❾
        print('Received from {}: {!r}'.format(client, query)) ❿
        if query:
            if ord(query[:1]) < 32: ⓫
                break
            lines = list(index.find_description_strs(query)) ⓬
            if lines:
                writer.writelines(line.encode() + CRLF for line in lines) ⓭
                writer.write(index.status(query, len(lines)).encode() + CRLF) ⓮
                yield from writer.drain() ⓯
                print('Sent {} results'.format(len(lines))) ⓰
    print('Close the client socket') ⓱
    writer.close() ⓲
```

- ❶ `UnicodeNameIndex` é a classe que cria o índice de nomes e oferece métodos de consulta.

- ❶ Quando instanciado, `UnicodeNameIndex` usa `charfinder_index.pickle`, se ele estiver disponível, ou cria-o, portanto a primeira execução pode demorar alguns segundos a mais para começar.¹⁰
- ❷ Essa é a corrotina que devemos passar para `asyncio.startserver`; os argumentos recebidos são um `asyncio.StreamReader` e um `asyncio.StreamWriter`.
- ❸ Esse laço mantém uma sessão que dura até um caractere de controle ser recebido do cliente.
- ❹ O método `StreamWriter.write` não é uma corrotina, apenas uma função comum; essa linha envia o prompt `?>`.
- ❺ `StreamWriter.drain` descarrega o buffer writer (faz um flush); é uma corrotina, portanto deve ser acionada com `yield from`.
- ❻ `StreamWriter.readline` é uma corrotina; ela devolve `bytes`.
- ❼ Um `UnicodeDecodeError` pode ocorrer quando o cliente Telnet envia caracteres de controle; se isso acontecer, devemos fingir que um caractere nulo foi enviado, para simplificar.
- ❽ Devolve o endereço remoto ao qual o socket está conectado.
- ❾ Faz log da consulta no console do servidor.
- ❿ Sai do laço se um caractere de controle ou um caractere nulo for recebido.
- ⓫ Devolve um gerador que produz strings com o codepoint Unicode, o caractere propriamente dito e seu nome (por exemplo, `U+0039\t9\tdIGIT NINE`); para simplificar, criei uma `list` a partir dele.
- ⓬ Envia as linhas (`lines`) convertidas em `bytes` usando a codificação default UTF-8, concatenando um carriage return e uma quebra de linha em cada uma; observe que o argumento é uma expressão geradora.
- ⓭ Escreve uma linha de status, por exemplo, `627 matches for 'digit'`.
- ⓮ Descarrega o buffer de saída.
- ⓯ Faz log da resposta no console do servidor.
- ⓰ Faz log do final da sessão no console do servidor.
- ⓱ Fecha `StreamWriter`.

¹⁰ Leonardo Rochael chamou a atenção para o fato de que a criação de `UnicodeNameIndex` poderia ser delegada para outra thread usando `loop.run_with_executor()` na função `main` do exemplo 18.15; assim o servidor estaria pronto para aceitar requisições imediatamente enquanto o índice fosse criado. Isso é verdade, mas consultar o índice é a única tarefa dessa aplicação, portanto isso não seria um ganho significativo. Porém fazer o que Leo sugeriu é um exercício interessante. Vá em frente e implemente isso se quiser.

A corrotina `handle_queries` tem um nome no plural porque ela inicia uma sessão interativa e trata várias consultas de cada cliente.

Observe que toda E/S no exemplo 18.14 está em bytes. Precisamos decodificar as strings recebidas da rede e codificar as strings enviadas. Em Python 3, a codificação default é UTF-8, e é essa codificação que estamos usando implicitamente.

Uma observação importante é que alguns dos métodos de E/S são corrotinas que devem ser acionadas com `yield from`, enquanto outras são funções simples. Por exemplo, `StreamWriter.write` é uma função comum, com base no pressuposto que, na maior parte do tempo, ela não é bloqueante porque escreve em um buffer. Por outro lado, `StreamWriter.drain`, que descarrega o buffer e executa a E/S propriamente dita, é uma corrotina, assim como `StreamReader.readline`. Enquanto escrevia este livro, uma melhoria importante na documentação da API de `asyncio` foi rotular claramente as corrotinas como tais.

O exemplo 18.15 lista a função `main` do módulo iniciado no exemplo 18.14.

Exemplo 18.15 – `tcp_charfinder.py` (continuação do exemplo 18.14): a função `main` cria e encerra o loop de eventos e o servidor de socket.

```
def main(address='127.0.0.1', port=2323): ❶
    port = int(port)
    loop = asyncio.get_event_loop()
    server_coro = asyncio.start_server(handle_queries, address, port,
                                      loop=loop) ❷
    server = loop.run_until_complete(server_coro) ❸
    host = server.sockets[0].getsockname() ❹
    print('Serving on {}'.format(host)) ❺
    try:
        loop.run_forever() ❻
    except KeyboardInterrupt: # CTRL+C pressionado
        pass

    print('Server shutting down.')
    server.close() ❼
    loop.run_until_complete(server.wait_closed()) ❽
    loop.close() ❾

if __name__ == '__main__':
    main(*sys.argv[1:]) ❿
```

❶ A função `main` pode ser chamada sem argumentos.

❷ Quando concluído, o objeto corrotina retornado por `asyncio.start_server` devolve uma instância de `asyncio.Server`, que é um servidor de socket TCP.

- ❸ Aciona `server_coro` para ativar o `server`.
- ❹ Obtém o endereço e a porta do primeiro `socket` do servidor e...
- ❺ ...exibe-os no console do servidor. Essa é a primeira saída gerada por esse script no console.
- ❻ Executa o loop de eventos; é aqui que `main` ficará bloqueado até ser finalizado quando CTRL-C for pressionado no console do servidor.
- ❼ Fecha o servidor.
- ❽ `server.wait_closed()` devolve um `future`; usamos `loop.run_until_complete` para deixar o `future` fazer o seu trabalho.
- ❾ Termina o loop de eventos.
- ❿ Esse é um atalho para tratar argumentos opcionais de linha de comando: expande `sys.argv[1:]` e passa esses dados para uma função `main` com argumentos `default` apropriados.

Observe como `run_until_complete` aceita uma corrotina (o resultado de `start_server`) ou um `Future` (o resultado de `server.wait_closed`). Se `run_until_complete` receber uma corrotina como argumento, ela será encapsulada em uma `Task`.

Talvez você ache mais fácil entender como o controle flui em `tcp_charfinder.py` se observar mais de perto a saída que ele gera no console do servidor, listada no exemplo 18.16.

Exemplo 18.16 – `tcp_charfinder.py`: esse é o lado do servidor da sessão representada na figura 18.2

```
$ python3 tcp_charfinder.py
Serving on ('127.0.0.1', 2323). Hit CTRL-C to stop. ❶
Received from ('127.0.0.1', 62910): 'chess black' ❷
Sent 6 results
Received from ('127.0.0.1', 62910): 'sun' ❸
Sent 10 results
Received from ('127.0.0.1', 62910): '\x00' ❹
Close the client socket ❺
```

- ❶ É a saída gerada por `main`.
- ❷ Primeira iteração do laço `while` em `handle_queries`.
- ❸ Segunda iteração do laço `while`.
- ❹ O usuário pressionou CTRL-C; o servidor recebe um caractere de controle e encerra a sessão.
- ❺ O `socket` do cliente é fechado, mas o servidor continua executando, pronto para servir outro cliente.

Observe como `main` exibe a mensagem `Serving on...` quase imediatamente e fica bloqueado na chamada a `loop.run_forever()`. Nesse ponto, o controle flui para o loop de eventos e permanece ali, voltando ocasionalmente para a corrotina `handle_queries`, que devolve o controle ao loop de eventos sempre que precisa esperar a rede à medida que dados são enviados ou recebidos. Enquanto o loop de eventos estiver ativo, uma nova instância da corrotina `handle_queries` será iniciada para cada cliente que se conectar ao servidor. Dessa maneira, muitos clientes podem ser tratados de forma concorrente por esse servidor simples. Isso continua até um `KeyboardInterrupt` ocorrer ou o sistema operacional matar o processo.

O código de `tcp_charfinder.py` aproveita a API de alto nível de Streams (<https://docs.python.org/3/library/asyncio-stream.html>) de `asyncio`, que oferece um servidor pronto para uso, de modo que você só precisa implementar uma função handler; essa função pode ser uma callback simples ou uma corrotina. Há também uma API de Transports e Protocols (<https://docs.python.org/3/library/asyncio-protocol.html>), de nível mais baixo, inspirada nas abstrações de transporte e protocolos do framework Twisted. Consulte a documentação de *Transports and Protocols* de `asyncio` (<http://bit.ly/1f6D9i6>) para obter mais informações, inclusive um servidor TCP de eco implementado com essa API de baixo nível.

A próxima seção apresenta um servidor HTTP para busca de caracteres.

Um servidor web com aiohttp

A biblioteca `aiohttp` que utilizamos nos exemplos das bandeiras com `asyncio` também suporta HTTP do lado do servidor, por isso usei-a para implementar o script `http_charfinder.py`. A figura 18.3 mostra a interface Web simples do servidor, exibindo o resultado de uma pesquisa a um emoji “cat face”.

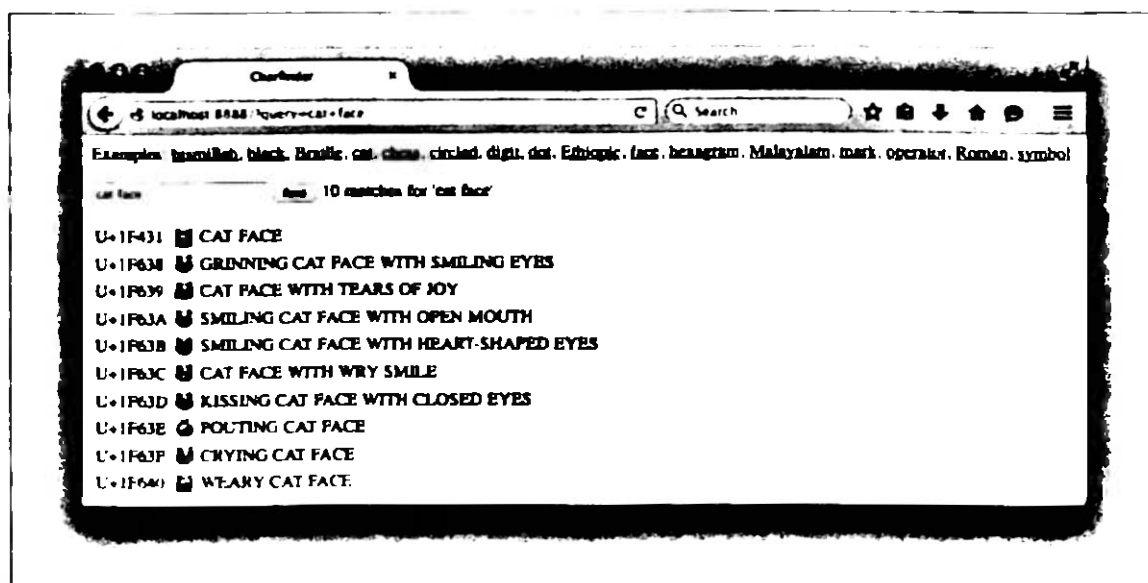


Figura 18.3 – Janela do navegador mostrando os resultados da pesquisa a “cat face” no servidor `http_charfinder.py`.



Alguns navegadores são melhores que outros para exibir Unicode. A captura de tela da figura 18.3 foi feita com Firefox no OS X, e obtive o mesmo resultado com o Safari. Porém navegadores Chrome e Opera atualizados no mesmo computador não exibiram caracteres de emoji como as carinhas de gatos. Outros resultados de pesquisa (por exemplo, “chess”) não apresentaram problemas, portanto é provável que seja um problema da fonte usada no Chrome e no Opera em OS X.

Começaremos analisando a parte mais interessante de *http_charfinder.py*: a metade inferior, em que o loop de eventos e o servidor HTTP são criados e encerrados. Veja o exemplo 18.17.

Exemplo 18.17 – *http_charfinder.py*: as funções *main* e *init*

```
@asyncio.coroutine
def init(loop, address, port): ❶
    app = web.Application(loop=loop) ❷
    app.router.add_route('GET', '/', home) ❸
    handler = app.make_handler() ❹
    server = yield from loop.create_server(handler,
                                           address, port) ❺
    return server.sockets[0].getsockname() ❻

def main(address="127.0.0.1", port=8888):
    port = int(port)
    loop = asyncio.get_event_loop()
    host = loop.run_until_complete(init(loop, address, port)) ❼
    print('Serving on {}'.format(host))
    try:
        loop.run_forever() ❺
    except KeyboardInterrupt: # CTRL+C pressionado
        pass
    print('Server shutting down.')
    loop.close() ❻

if __name__ == '__main__':
    main(*sys.argv[1:])
```

- ❶ A corrotina *init* produz um servidor para o loop de eventos acionar.
- ❷ A classe *aiohttp.web.Application* representa uma aplicação web...
- ❸ ... com rotas que mapeiam padrões de URL a funções de tratamento; aqui GET / é encaminhado para a função *home* (veja o exemplo 18.18).

- ❶ O método `app.make_handler` devolve uma instância de `aihttp.web.RequestHandler` para tratar requisições HTTP de acordo com as rotas configuradas no objeto `app`.
- ❷ `create_server` ativa o servidor, usando `handler` como o handler de protocolo e associando-o a `address` e `port`.
- ❸ Devolve o endereço e a porta do primeiro socket do servidor.
- ❹ Executa `init` para iniciar o servidor e obtém seu endereço e a porta.
- ❺ Executa o loop de eventos; `main` ficará bloqueado aqui, enquanto o loop de eventos estiver no controle.
- ❻ Encerra o loop de eventos.

Para conhecer melhor a API de `asyncio`, é interessante comparar como os servidores são criados no exemplo 18.17 e no exemplo com TCP (Exemplo 18.15), mostrado antes.

No exemplo anterior com TCP, o servidor foi criado e escalonado para executar na função `main` com as duas linhas a seguir:

```
server_coro = asyncio.start_server(handle_queries, address, port,  
                                   loop=loop)  
server = loop.run_until_complete(server_coro)
```

No exemplo com HTTP, a função `init` cria o servidor assim:

```
server = yield from loop.create_server(handler,  
                                       address, port)
```

Mas `init` é uma corrotina, e o que a faz rodar é a função `main`, com esta linha:

```
host = loop.run_until_complete(init(loop, address, port))
```

Tanto `asyncio.start_server` quanto `loop.create_server` são corrotinas que devolvem objetos `asyncio.Server`. Para iniciar um servidor e devolver uma referência a ele, cada uma dessas corrotinas deve executar até o fim. No exemplo com TCP, isso foi feito chamando `loop.run_until_complete(server_coro)`, em que `server_coro` era o resultado de `asyncio.start_server`. No exemplo com HTTP, `create_server` é chamado em uma expressão `yield from` na corrotina `init`, que, por sua vez, é acionada pela função `main` quando ela chama `loop.run_until_complete(init(...))`.

Menciono isso para enfatizar um fato essencial que discutimos antes: uma corrotina só faz algo quando é acionada, e, para acionar uma `asyncio.coroutine`, você precisa usar `yield from` ou passá-la para uma das várias funções de `asyncio` que aceitem corrotina ou futuros como argumento, por exemplo, `run_until_complete`.

O exemplo 18.18 mostra a função `home`, configurada para tratar o URL / (raiz) em nosso servidor HTTP.

Exemplo 18.18 – `http_charfinder.py`: a função `home`

```
def home(request): ❶
    query = request.GET.get('query', '').strip() ❷
    print('Query: {!r}'.format(query)) ❸
    if query: ❹
        descriptions = list(index.find_descriptions(query))
        res = '\n'.join(ROW_TPL.format(**vars(descr))
                        for descr in descriptions)
        msg = index.status(query, len(descriptions))
    else:
        descriptions = []
        res = ''
        msg = 'Enter words describing characters.'
    html = template.format(query=query, result=res, ❺
                           message=msg)
    print('Sending {} results'.format(len(descriptions))) ❻
    return web.Response(content_type=CONTENT_TYPE, text=html) ❼
```

- ❶ Um handler de rotas recebe uma instância de `aioshttp.web.Request`.
- ❷ Obtém a string de query sem os caracteres em branco no início e no fim.
- ❸ Faz log da consulta no console do servidor.
- ❹ Se houver uma consulta, associa `res` às linhas da tabela HTML produzidas a partir do resultado da consulta a `index` e associa `msg` a uma mensagem de status.
- ❺ Renderiza a página HTML.
- ❻ Faz log da resposta no console do servidor.
- ❼ Constrói `Response` e a devolve.

Observe que `home` não é uma corrotina, e não precisa ser se não tiver expressões `yield from`. A documentação do método `add_route` (<http://bit.ly/1HGu5dz>) em `aioshttp` afirma que o handler “é convertido internamente em uma corrotina se for uma função comum”.

Há uma desvantagem na simplicidade da função `home` do exemplo 18.18. O fato de ser uma função comum e não uma corrotina é um sintoma de um problema maior: a necessidade de repensar como implementamos aplicações web para ter um alto nível de concorrência. Vamos considerar essa questão.

Clientes mais inteligentes para melhorar a concorrência

A função `home` do exemplo 18.18 é muito semelhante a uma função de `view` em Django ou Flask. Não há nada de assíncrono em sua implementação: ela recebe uma requisição, busca dados em um banco de dados e constrói uma resposta renderizando uma página HTML completa. Nesse exemplo, o “banco de dados” é o objeto `UnicodeNameIndex`, que está na memória. Mas acessar um banco de dados de verdade deveria ser uma operação assíncrona; caso contrário, você bloqueará o loop de eventos enquanto espera o resultado do banco de dados. Por exemplo, o pacote `aiopg` (<https://aiopg.readthedocs.org/en/stable/>) oferece um driver assíncrono para PostgreSQL compatível com `asyncio`; ele permite que você use `yield from` para enviar consultas e buscar resultados, de modo que sua função de `view` pode se comportar apropriadamente como uma corrotina.

Além de evitar chamadas bloqueantes, sistemas com muita concorrência devem separar porções grandes de tarefas em partes menores para permanecer responsivos. O servidor `http_charfinder.py` ilustra essa questão: se procurar “`cjk`”, você receberá 75.821 ideogramas chineses, japoneses e coreanos.¹¹ Nesse caso, a função `home` devolverá um documento HTML de 53 MB com uma tabela de 75.821 linhas.

Em meu computador, foram necessários dois segundos para buscar a resposta da consulta a “`cjk`”, usando o cliente HTTP `curl` de linha de comando, de um servidor `http_charfinder.py` local. Um navegador demora mais ainda para montar a página com uma tabela tão grande assim. É claro que a maioria das consultas devolve respostas muito menores: uma consulta a “`braille`” devolve 256 linhas em uma página de 19 KB e demora 0,017s em meu computador. Mas se o servidor gasta 2s servindo uma única consulta a “`cjk`”, todos os demais clientes ficarão esperando pelo menos 2s, o que é inaceitável.

A maneira de evitar o problema da resposta longa é implementar paginação: devolver resultados, digamos, com no máximo duzentas linhas e fazer o usuário clicar ou fazer rolar a página para buscar mais dados. Se você der uma olhada no módulo `charfinder.py` no repositório de código de *Python fluente* (<http://bit.ly/1JItSti>), verá que o método `UnicodeNameIndex.find_descriptions` aceita argumentos `start` e `stop` opcionais: são offsets para suporte à paginação. Então você poderia devolver os primeiros duzentos resultados e usar AJAX ou até mesmo WebSockets para enviar o próximo lote quando – e se – o usuário quiser vê-lo.

A maior parte do código necessário para enviar resultados em lotes estaria no navegador. Isso explica por que o Google e todos os serviços de larga escala na Internet

¹¹ É isso que CJK quer dizer: caracteres chineses, japoneses e coreanos (Chinese, Japanese and Korean), que estão continuamente em expansão. Versões futuras de Python poderão suportar mais ideogramas CJK que Python 3.4.

contam com muito código do lado do cliente para desenvolver seus serviços: clientes assíncronos inteligentes fazem melhor uso dos recursos do servidor.

Embora clientes inteligentes possam ajudar até mesmo aplicações Django que usem um estilo mais antigo, para realmente servi-los bem, precisamos de frameworks que tenham suporte à programação assíncrona do início ao fim: do tratamento de requisições e respostas HTTP até o acesso a banco de dados. Isso vale, em especial, se você quiser implementar serviços de tempo real como jogos e streaming de mídias com WebSockets.¹²

Melhorar `http_charfinder.py` para que suporte download progressivo fica como exercício para o leitor. Você ganhará pontos extras se implementar “rolagem infinita” (infinite scroll) como faz o Twitter. Com esse desafio, encerro nossa discussão sobre programação concorrente com `asyncio`.

Resumo do capítulo

Este capítulo apresentou uma maneira totalmente nova de implementar concorrência em Python, aproveitando `yield from`, corrotinas, futures e o loop de eventos de `asyncio`. Os primeiros exemplos simples – os scripts com a barra giratória – foram projetados para mostrar uma comparação lado a lado das abordagens de concorrência em `threading` e `asyncio`.

Então discutimos as especificidades de `asyncio.Future`, enfocando seu suporte a `yield from` e seu relacionamento com corrotinas e `asyncio.Task`. Em seguida, analisamos o script para download de bandeiras baseado em `asyncio`.

Refletimos então sobre os números de Ryan Dahl para latência de E/S e o efeito de chamadas bloqueantes. Para manter um programa vivo, apesar das funções bloqueantes inevitáveis, há duas soluções: usar threads ou chamadas assíncronas – a última opção, implementada com callbacks ou corrotinas.

Na prática, bibliotecas assíncronas dependem de threads de mais baixo nível para funcionar – até threads no nível do kernel –, mas o usuário da biblioteca não cria threads nem precisa estar ciente de seu uso na infraestrutura. No âmbito da aplicação, simplesmente garantimos que nenhum código nosso seja bloqueante, e o loop de eventos cuida internamente da concorrência. Evitar o overhead das threads de nível de usuário é o principal motivo pelo qual sistemas assíncronos podem gerenciar mais conexões concorrentes que sistemas com várias threads (multithreaded).

Retomando os exemplos de download de bandeiras, o acréscimo de uma barra de progresso e um tratamento de erros adequado exigiram uma refatoração significativa, em particular com a mudança de `asyncio.wait` para `asyncio.as_completed`, que nos

¹² Tenho mais a dizer sobre essa tendência na seção “Ponto de vista” na página 640.

forçou a transferir a maior parte das funcionalidades de `download_many` para uma nova corrotina `downloader_coro` de modo que pudéssemos usar `yield from` para obter os resultados dos futures produzidos por `asyncio.as_completed`, um por um.

Então vimos como delegar tarefas bloqueantes – por exemplo, salvar um arquivo – a um pool de threads usando o método `loop.run_in_executor`.

Seguiu-se então uma discussão sobre como corrotinas resolvem os principais problemas das callbacks: perda de contexto ao executar tarefas assíncronas de vários passos e falta de um contexto apropriado para tratamento de erros.

O próximo exemplo – buscar os nomes dos países, juntamente com as imagens das bandeiras – mostrou como a combinação entre corrotinas e `yield from` evita o chamado inferno de callbacks (callback hell). Um procedimento de vários passos fazendo chamadas assíncronas com `yield from` se parece com um código sequencial se não prestarmos atenção às palavras reservadas `yield from`.

Os últimos exemplos do capítulo foram os servidores TCP e HTTP com `asyncio`, que permitem pesquisar caracteres Unicode pelo nome. A análise do servidor HTTP terminou com uma discussão sobre a importância do JavaScript do lado cliente para suporte a mais concorrência no lado do servidor, permitindo que o cliente faça requisições menores por demanda, em vez de fazer download de páginas HTML grandes.

Leituras complementares

Nick Coghlan, um core developer de Python, fez o seguinte comentário na versão preliminar da PEP 3156 – *Asynchronous IO Support Rebooted: the “asyncio” Module* (Suporte renovado a E/S assíncrona: o módulo “asyncio”, <http://bit.ly/1HGuPPE>) em janeiro de 2013:

Em algum ponto no início da PEP, talvez seja necessária uma descrição concisa de duas APIs para esperar um future assíncrono:

1. `f.add_done_callback(...)`
2. `yield from f` em uma corrotina (retomará a corrotina quando o future terminar, com o resultado ou a exceção, conforme apropriado)

No momento, elas estão escondidas em APIs muito maiores, apesar de serem fundamentais para entender como tudo que está acima da camada do loop de eventos central interage.¹³

Guido van Rossum, autor da PEP-3156 (<https://www.python.org/dev/peps/pep-3156/>), não seguiu o conselho de Coghlan. Começando pela PEP-3156, a documentação

¹³ Comentário sobre a PEP-3156 em uma mensagem de 20 de janeiro de 2013 (<http://bit.ly/1f6DGRI>) na lista `python-ideas`.

de `asyncio` é bastante detalhada, mas não é muito amigável. Os nove arquivos `.rst` que compõem a documentação do pacote `asyncio` (<http://bit.ly/1HGuuwq>) totalizam 128 KB – são aproximadamente 71 páginas. Na biblioteca-padrão, somente o capítulo “Built-in Types” (Tipos embutidos, <http://bit.ly/1HGurAX>) é maior, e ele inclui a API dos tipos numéricos, tipos de sequência, geradores, mapeamentos, conjuntos, `bool`, gerenciadores de contexto etc.

A maior parte das páginas do manual de `asyncio` enfoca conceitos e a API. Há diagramas úteis e exemplos espalhados em todos os lugares, mas uma seção bem prática é “18.5.11. Develop with asyncio” (Desenvolver com `asyncio`, <https://docs.python.org/3/library/asyncio-dev.html>), que apresenta padrões de uso essenciais. A documentação de `asyncio` precisa de mais conteúdo para explicar como a API deve ser usada.

Por ser bem novo, faltam publicações que discutam o pacote `asyncio`. O livro *Parallel Programming with Python* (Packt, 2014) de Jan Palach é o único livro que encontrei com um capítulo sobre `asyncio`, mas é um capítulo pequeno.

No entanto há apresentações excelentes sobre `asyncio`. A melhor que encontrei é “Fan-In and Fan-Out: The Crucial Components of Concurrency” (Fan-in e fan-out: os componentes cruciais da concorrência, <http://bit.ly/1f6DIZo>) de Brett Slatkin, cujo subtítulo é “Why do we need Tulip? (a.k.a., PEP 3156 – asyncio)” [Por que precisamos de Tulip? (também conhecido como PEP-3156 – asyncio)], que ele apresentou na PyCon 2014 em Montreal (vídeo em <http://bit.ly/1HGurY2>). Em 30 minutos, Slatkin mostra um exemplo simples de web crawler, destacando como `asyncio` deve ser usado. Guido van Rossum está na plateia e menciona que ele também escreveu um web crawler como exemplo para motivação ao uso de `asyncio`; o código de Guido (<http://bit.ly/1HGub4K>) não depende de `aiohttp` – ele usa somente a biblioteca-padrão. Slatkin também escreveu o post esclarecedor: “Python’s asyncio Is for Composition, Not Raw Performance” (`asyncio` de Python é para composição, e não para desempenho, <http://bit.ly/1f6Djwj>).

Outras palestras sobre `asyncio` que não podem deixar de ser vistas são do próprio Guido van Rossum: a apresentação na PyCon US 2013 (<http://bit.ly/1HGueh0>) e as palestras no LinkedIn (<http://bit.ly/1HGudd0>) e no Twitter University (<http://bit.ly/1HGuexy>). Também recomendo ver “A Deep Dive into PEP-3156 and the New asyncio Module” (Um mergulho profundo na PEP-3156 e o novo módulo `asyncio`, slides em <http://bit.ly/1HGuf4D>, vídeo em <http://bit.ly/1HGufBq>) de Saúl Ibarra Corretgé.

Dino Viehland mostrou como `asyncio` pode ser integrado ao loop de eventos de Tkinter em sua palestra “Using futures for async GUI programming in Python 3.3” (Usando futures para programação assíncrona de GUI em Python 3.3, <http://bit.ly/1HGuoos>) na PyCon US 2013. Viehland mostra como é fácil implementar as partes essenciais da interface `asyncio.AbstractEventLoop` sobre outro loop de eventos. Seu código foi escrito com Tulip, antes da adição de `asyncio` à biblioteca-padrão; adaptei-o para que

funcionasse com a versão de `asyncio` para Python 3.4. Minha refatoração atualizada está no GitHub (<http://bit.ly/1HGulck>).

Victor Stinner – um colaborador do núcleo de `asyncio` e autor do pacote `Trollius` para retrocompatibilidade (<http://trollius.readthedocs.org/>) – atualiza regularmente uma lista de links relevantes: *The new Python asyncio module aka “tulip”* (O novo módulo `asyncio` de Python, também conhecido como “tulip”, <http://bit.ly/1HGumwZ>).

Outras coletâneas de materiais sobre `asyncio` estão em `Asyncio.org` (<http://asyncio.org/>) e em `aio-lib`s (<https://github.com/aio-lib>) no Github, onde você encontrará drivers assíncronos para PostgreSQL, MySQL e vários bancos de dados NoSQL. Não testei esses drivers, mas os projetos pareciam estar bem ativos quando escrevi este livro.

Os web services serão um caso de uso importante para `asyncio`. Seu código provavelmente dependerá da biblioteca `aiohttp` (<http://aiohttp.readthedocs.org/en/>) conduzida por Andrew Svetlov. Você também vai querer configurar um ambiente para testar seu código de tratamento de erro, e o “caos TCP proxy” `Vaurien` (<http://vaurien.readthedocs.org/en/1.8/>), projetado por Alexis Métaireau e Tarek Ziadé, é muito conveniente para isso. `Vaurien` foi criado para o projeto Mozilla Services (<https://mozilla-services.github.io/>) e permite que você introduza atrasos e erros aleatórios em tráfego TCP entre seu programa e servidores no backend, como bancos de dados e provedores de web services.

Ponto de vista

O “Um Laço”

Por muito tempo, a programação assíncrona tem sido a abordagem preferida pela maioria dos pythonistas para aplicações de rede, mas sempre havia o dilema de escolher uma das várias bibliotecas mutuamente incompatíveis. Ryan Dahl menciona `Twisted` como fonte de inspiração para `Node.js`, e `Tornado` popularizou o uso de corrotinas para programação orientada a eventos em Python.

No mundo de JavaScript, existem debates entre os que defendem `callbacks` simples e os que propõem várias abstrações de nível mais alto que competem entre si. Versões iniciais da API de `Node.js` usavam `Promises` – semelhantes aos nossos `Futures` –, mas Ryan Dahl decidiu padronizar somente com base em `callbacks`. James Coglan argumenta que essa foi a maior oportunidade perdida pelo `Node` (<http://bit.ly/1xNcNHZ>).

Em Python, o debate terminou: a adição de `asyncio` à biblioteca-padrão estabelece corrotinas e `futures` como o modo pythônico de implementar código assíncrono. Além do mais, o pacote `asyncio` define interfaces-padrões para `futures` assíncronos e o loop de eventos, oferecendo implementações de referência para eles.

O *Zen do Python* aplica-se perfeitamente:

Deve haver um – e, de preferência, apenas um – modo óbvio de fazer algo.

Embora esse modo possa não ser óbvio à primeira vista, a menos que você seja holandês.

Talvez seja necessário um passaporte holandês para achar `yield from` óbvio. Para este brasileiro, não era óbvio no início, mas, depois de um tempo, peguei o jeito. Mais importante ainda, `asyncio` foi projetado para que seu loop de eventos possa ser substituído por um pacote externo. É por isso que as funções `asyncio.get_event_loop` e `asyncio.set_event_loop` existem; elas fazem parte de uma API abstrata *Event Loop Policy* (Política do loop de eventos, <http://bit.ly/1HGuUTy>).

O Tornado já tem uma classe `AsyncIOMainLoop` (<http://tornado.readthedocs.org/en/latest/asyncio.html>) que implementa a interface `asyncio.AbstractEventLoop`, de modo que você pode executar código assíncrono usando ambas as bibliotecas no mesmo loop de eventos. Temos também o curioso projeto Quamash (<https://pypi.python.org/pypi/Quamash/>), que integra `asyncio` ao loop de eventos de Qt para desenvolver aplicações GUI com PyQt ou PySide. Esses são apenas dois pacotes orientados a eventos que permitem interoperação, entre um número crescente deles, viáveis por causa de `asyncio`.

Clientes HTTP mais inteligentes, como aplicações web single-page (como o Gmail) ou aplicativos de smartphone, exigem respostas rápidas e leves, e que o servidor possa enviar atualizações sem que o cliente tenha de pedir. Essas necessidades são mais bem servidas por frameworks assíncronos em vez de frameworks Web tradicionais como Django, projetados para servir páginas HTML totalmente renderizadas e que não têm suporte para acesso assíncrono a bancos de dados.

O protocolo WebSockets foi projetado para permitir atualizações em tempo real para clientes que estejam sempre conectados, desde jogos a aplicações de streaming. Isso exige servidores assíncronos altamente concorrentes, capazes de manter interações contínuas com centenas ou milhares de clientes. A arquitetura de `asyncio` tem suporte muito bom para WebSockets, e pelo menos duas bibliotecas já o implementam com base em `asyncio`: Autobahn|Python (<http://autobahn.ws/python/>) e WebSockets (<http://aagustin.github.io/websockets/>).

Essa tendência geral – apelidada de “Web em tempo real” (the real-time Web) – é um fator-chave na demanda por Node.js e o motivo pelo qual reunir forças em torno de `asyncio` é tão importante para o ecossistema de Python. Ainda há muito trabalho a fazer. Para começar, precisamos de APIs para servidor e cliente HTTP assíncronos na biblioteca-padrão, um DBAPI 3.0 (<http://bit.ly/1HGuVGY>) assíncrono e novos drivers de banco de dados com base em `asyncio`.

A principal vantagem de Python 3.4 com `asyncio` em relação a Node.js é Python: uma linguagem com design melhor, com corrotinas e `yield from` para implementar código assíncrono mais fácil de manter que as callbacks primitivas de JavaScript. Nossa principal desvantagem são as bibliotecas: Python vem com “pilhas incluídas” (“batteries included”), mas nossas pilhas não foram projetadas para programação assíncrona. O rico ecossistema de bibliotecas de Node.js é totalmente desenvolvido em torno de chamadas assíncronas. Mas Python e Node.js têm um problema que Go e Erlang resolveram desde o princípio: não temos uma maneira transparente de codar aproveitando todos os núcleos de CPU disponíveis.

Padronizar a interface do loop de eventos e uma biblioteca assíncrona foi uma cartada importante, e somente nosso BDFL poderia ter feito isso acontecer, considerando que havia alternativas bastante enraizadas e de alta qualidade disponíveis. Ele fez isso após consultar autores dos principais frameworks Python assíncronos. A influência de Glyph Lefkowitz, líder do Twisted, é a mais evidente. O post “Deconstructing Deferred” (Desconstruindo deferred, <http://bit.ly/1HGUXPa>) de Guido no grupo Python-tulip é leitura obrigatória se você quiser entender por que `asyncio.Future` é diferente da classe `Deferred` de Twisted. Deixando claro seu respeito pelo framework assíncrono maior e mais antigo de Python, Guido também criou o meme WWTD – What Would Twisted Do? (O que Twisted faria?) – quando discutia opções de design no grupo python-twisted.¹⁴

Felizmente, Guido van Rossum conduziu a comunidade de uma forma que deixou Python mais bem posicionado para encarar os desafios atuais da concorrência. Dominar `asyncio` exige esforços. Mas, se você planeja escrever aplicações de rede concorrentes em Python, procure o “Um Laço”.

Um Laço para a todos governar, Um Laço para encontrá-los.

Um Laço para todos trazer e na vivacidade uni-los.

¹⁴ Veja a mensagem de Guido de 29 de janeiro de 2015 (<http://bit.ly/1f6E2qT>), seguida imediatamente de uma resposta de Glyph.

PARTE VI

Metaprogramação

CAPÍTULO 19

Atributos dinâmicos e propriedades

A importância crucial das propriedades é que sua existência torna totalmente seguro e, na verdade, aconselhável expor atributos de dados públicos como parte da interface pública de sua classe.¹

— Alex Martelli

Colaborador de Python e autor de livros sobre essa linguagem

Atributos de dados e métodos são coletivamente conhecidos como *atributos* em Python: um método é apenas um atributo *invocável* (callable). Além de atributos de dados e métodos, também podemos criar propriedades que podem ser usadas para substituir um atributo de dado público por *métodos de acesso* (ou seja, getter/setter), sem alterar a interface da classe. Isso está de acordo com o *Princípio do acesso uniforme*:

Todos os serviços oferecidos por um módulo devem estar disponíveis por meio de uma notação uniforme, que não revele o fato de estarem implementados por armazenamento ou processamento.²

Além de propriedades, Python oferece uma API rica para controlar acesso a atributos e implementar atributos dinâmicos. O interpretador chama métodos especiais como `__getattr__` e `__setattr__` para avaliar o acesso a atributos que usam notação de ponto (por exemplo, `obj.attr`). Uma classe definida pelo usuário que implemente `__getattr__` pode implementar “atributos virtuais”, computando valores durante a execução sempre que alguém tenta ler um atributo inexistente como `obj.atributo_que_nao_existe`.

Implementar atributos dinâmicos é o tipo de metaprogramação feito por autores de framework. No entanto, em Python, as técnicas básicas são tão simples que qualquer pessoa pode colocá-las em prática, mesmo que seja para pequenas tarefas do cotidiano. É assim que começaremos este capítulo.

¹ Alex Martelli, *Python in a Nutshell*, 2E (O'Reilly, <http://shop.oreilly.com/product/9780596100469.do>), p. 101.

² Bertrand Meyer, *Object-Oriented Software Construction*, 2E, p. 57.

Processando dados com atributos dinâmicos

Nos próximos exemplos, aproveitaremos atributos dinâmicos para trabalhar com um feed de dados JSON publicado pela O'Reilly para a conferência OSCON de 2014. O exemplo 19.1 mostra quatro registros desse feed de dados.³

Exemplo 19.1 – Exemplos de registros de `osconfeed.json`; os conteúdos de alguns campos estão abreviados

```
{ "Schedule":
  { "conferences": [{"serial": 115 }],
    "events": [
      { "serial": 34505,
        "name": "Why Schools Don't Use Open Source to Teach Programming",
        "event_type": "40-minute conference session",
        "time_start": "2014-07-23 11:30:00",
        "time_stop": "2014-07-23 12:10:00",
        "venue_serial": 1462,
        "description": "Aside from the fact that high school programming...",
        "website_url": "http://oscon.com/oscon2014/public/schedule/detail/34505",
        "speakers": [157509],
        "categories": ["Education"] }
    ],
    "speakers": [
      { "serial": 157509,
        "name": "Robert Lefkowitz",
        "photo": null,
        "url": "http://sharewave.com/",
        "position": "CTO",
        "affiliation": "Sharewave",
        "twitter": "sharewaveteam",
        "bio": "Robert 'r0ml' Lefkowitz is the CTO at Sharewave, a startup..." }
    ],
    "venues": [
      { "serial": 1462,
        "name": "F151",
        "category": "Conference Venues" }
    ]
  }
}
```

³ Você pode ler sobre esse feed e as regras para usá-lo em “DIY: OSCON schedule” (Faça você mesmo: agenda da OSCON, <http://bit.ly/1TxUXBP>). O arquivo JSON original com 744 KB ainda estava online (<http://www.oreilly.com/pub/sc/osconfeed>) quando escrevi este capítulo. Uma cópia chamada *osconfeed.json* pode ser encontrada no diretório *oscon-schedule/data/* no repositório de código de Python *fluent* (<http://bit.ly/1TxUXBP>).

O exemplo 19.1 mostra 4 dos 895 registros do feed JSON. Como podemos ver, todo o conjunto de dados é um único objeto JSON com a chave "Schedule", e seu valor é outro mapeamento com quatro chaves: "conferences", "events", "speakers" e "venues". Cada uma dessas quatro chaves está associada a uma lista de registros. No exemplo 19.1, cada lista tem um registro, mas, no conjunto de dados completo, essas listas têm dezenas ou centenas de registros – com exceção de "conferences", que armazena apenas o único registro mostrado. Todos os itens nessas quatro listas têm um campo "serial", que é um identificador único na lista.

O primeiro script que escrevi para lidar com o feed da OSCON simplesmente faz download do feed, evitando tráfego desnecessário verificando se há uma cópia local. Isso faz sentido, pois a OSCON 2014 agora é passado, portanto esse feed não será mais atualizado.

Não há metaprogramação no exemplo 19.2; praticamente tudo se reduz a esta expressão: `json.load(fp)` – mas é suficiente para nos permitir explorar o conjunto de dados. A função `osconfeed.load` será usada nos próximos exemplos.

Exemplo 19.2 – `osconfeed.py`: fazendo download de `osconfeed.json` (doctests estão no exemplo 19.3)

```
from urllib.request import urlopen
import warnings
import os
import json

URL = 'http://www.oreilly.com/pub/sc/osconfeed'
JSON = 'data/osconfeed.json'

def load():
    if not os.path.exists(JSON):
        msg = 'downloading {} to {}'.format(URL, JSON)
        warnings.warn(msg) ❶
        with urlopen(URL) as remote, open(JSON, 'wb') as local: ❷
            local.write(remote.read())

    with open(JSON) as fp:
        return json.load(fp) ❸
```

- ❶ Gera um warning se um novo download vai ser feito.
- ❷ `with` usando dois gerenciadores de contexto (permitido desde Python 2.7 e 3.1) para ler o arquivo remoto e salvá-lo.
- ❸ A função `json.load` faz parse de um arquivo JSON e devolve objetos Python nativos. Nesse feed, temos os seguintes tipos: `dict`, `list`, `str` e `int`.

Com o código do exemplo 19.2, podemos inspecionar qualquer campo dos dados. Veja o exemplo 19.3.

Exemplo 19.3 – osconfeed.py: doctests para o exemplo 19.2

```
>>> feed = load() ❶
>>> sorted(feed['Schedule'].keys()) ❷
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed['Schedule'].items()):
...     print('{:3} {}'.format(len(value), key)) ❸
...
   1 conferences
 484 events
 357 speakers
   53 venues
>>> feed['Schedule']['speakers'][-1]['name'] ❹
'Carina C. Zona'
>>> feed['Schedule']['speakers'][-1]['serial'] ❺
141598
>>> feed['Schedule']['events'][40]['name']
'There *Will* Be Bugs'
>>> feed['Schedule']['events'][40]['speakers'] ❻
[3471, 5199]
```

- ❶ feed é um dict que armazena dicionários e listas aninhados, com valores string e inteiros.
- ❷ Lista as quatro coleções de registros em "Schedule".
- ❸ Exibe o número de registros em cada coleção.
- ❹ Navega pelos dicionários e listas aninhados para obter o nome do último palestrante.
- ❺ Obtém o número identificador desse mesmo palestrante.
- ❻ Cada evento tem uma lista de 'speakers' com 0 ou mais números identificadores dos palestrantes.

Explorando dados JSON ou similares com atributos dinâmicos

O exemplo 19.2 é simples, mas a sintaxe `feed['Schedule']['events'][40]['name']` não é muito prática. Em JavaScript, você pode obter o mesmo valor escrevendo `feed.Schedule.events[40].name`. É fácil implementar uma classe similar a dict que faça o mesmo em

Python – há muitas implementações na Web.⁴ Implementei minha própria classe FrozenJSON, mais simples que a maioria das receitas, pois tem suporte apenas para leitura: serve só para explorar os dados. No entanto é também recursiva, lidando automaticamente com mapeamentos e listas aninhados.

O exemplo 19.4 apresenta uma demonstração de FrozenJSON, e o código-fonte está no exemplo 19.5.

Exemplo 19.4 – FrozenJSON no exemplo 19.5 permite ler atributos como `name` e chamar métodos como `.keys()` e `.items()`

```
>>> from osconfeed import load
>>> raw_feed = load()
>>> feed = FrozenJSON(raw_feed) ❶
>>> len(feed.Schedule.speakers) ❷
357
>>> sorted(feed.Schedule.keys()) ❸
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed.Schedule.items()): ❹
...     print('{:3} {}'.format(len(value), key))
...
    1 conferences
484 events
357 speakers
    53 venues
>>> feed.Schedule.speakers[-1].name ❺
'Carina C. Zona'
>>> talk = feed.Schedule.events[40]
>>> type(talk) ❻
<class 'explore0.FrozenJSON'>
>>> talk.name
'There *Will* Be Bugs'
>>> talk.speakers ❼
[3471, 5199]
>>> talk.flavor ❸
Traceback (most recent call last):
...
KeyError: 'flavor'
```

- ❶ Cria uma instância de FrozenJSON a partir de `raw_feed` composto de dicionários e listas aninhados.

⁴ Uma implementação citada com frequência é `AttrDict` (<https://pypi.python.org/pypi/attrdict>); outra que permite uma criação rápida de mapeamentos aninhados é `addict` (<https://pypi.python.org/pypi/addict>).

- ❷ FrozenJSON permite percorrer dicionários aninhados usando notação de atributo; aqui mostramos o tamanho da lista de palestrantes.
- ❸ Métodos dos dicionários subjacentes também podem ser acessados, como `.keys()`, para recuperar os nomes das coleções de registros.
- ❹ Usando `items()`, podemos recuperar os nomes das coleções de registros e seus conteúdos para exibir o `len()` de cada um.
- ❺ Uma `list`, como `feed.Schedule.speakers`, continua sendo uma lista, mas os itens nela contidos são convertidos para FrozenJSON se forem mapeamentos.
- ❻ O item 40 na lista `events` era um `dict`; agora é uma instância de FrozenJSON.
- ❼ Registros de eventos têm uma lista `speakers` com os números identificadores dos palestrantes.
- ❽ Tentar ler um atributo ausente levanta `KeyError`, em vez de levantar o usual `AttributeError`.

A pedra angular da classe FrozenJSON é o método `__getattr__`, que já usamos no exemplo de Vector na seção “Vector tomada #3: acesso dinâmico a atributos” na página 328 para recuperar componentes de Vector usando uma letra – `v.x`, `v.y`, `v.z` etc. É essencial lembrar que o método especial `__getattr__` é chamado pelo interpretador somente quando o processo normal não consegue recuperar um atributo (ou seja, quando o atributo nomeado não é encontrado na instância, nem na classe ou em suas superclasses).

A última linha do exemplo 19.4 expõe um pequeno problema com a implementação: idealmente, tentar ler um atributo ausente deve levantar `AttributeError`. Na verdade, implementei o tratamento de erro, mas o tamanho do método `__getattr__` dobrou e desviava a atenção da lógica mais importante que eu queria mostrar, portanto deixei-o de fora por motivos didáticos.

Como mostra o exemplo 19.5, a classe FrozenJSON tem apenas dois métodos (`__init__`, `__getattr__`) e um atributo de instância `__data`, portanto tentativas de recuperar um atributo de qualquer nome diferente desses fará `__getattr__` ser disparado. Esse método inicialmente verificará se o `dict self.__data` tem um atributo (não uma chave!) com esse nome; isso permite às instâncias de FrozenJSON tratar qualquer método de `dict`, por exemplo, `items`, delegando para `self.__data.items()`. Se `self.__data` não tiver um atributo com o nome especificado, `__getattr__` usa `name` como chave para recuperar um item de `self.__dict` e passa esse item para `FrozenJSON.build`. Isso permite navegar por estruturas aninhadas nos dados JSON, pois cada mapeamento aninhado é convertido para outra instância de FrozenJSON pelo método de classe `build`.

Exemplo 19.5 – explore0.py: transforma um conjunto de dados JSON em um FrozenJSON que armazena objetos FrozenJSON aninhados, listas e tipos simples

```
from collections import abc
```

```
class FrozenJSON:
```

```
    """Uma fachada somente de leitura para navegar por um objeto JSON ou similar
       usando notação de atributo
    """
```

```
    def __init__(self, mapping):
        self.__data = dict(mapping) ❶
```

```
    def __getattr__(self, name): ❷
        if hasattr(self.__data, name):
            return getattr(self.__data, name) ❸
        else:
            return FrozenJSON.build(self.__data[name]) ❹
```

```
    @classmethod
```

```
    def build(cls, obj): ❺
        if isinstance(obj, abc.Mapping): ❻
            return cls(obj)
        elif isinstance(obj, abc.MutableSequence): ❼
            return [cls.build(item) for item in obj]
        else: ❽
            return obj
```

- ❶ Constrói um dict a partir do argumento `mapping`. Isso tem dois propósitos: garante que recebemos um dict (ou algo que possa ser convertido em um dict) e cria uma cópia local por segurança.
- ❷ `__getattr__` é chamado somente quando não há um atributo com esse `name`.
- ❸ Se `name` for igual a um atributo de `__data`, devolve-o. É assim que chamadas a métodos como `keys` são tratadas.
- ❹ Caso contrário, busca o item com a chave `name` em `self.__data` e devolve o resultado da chamada a `FrozenJSON.build()` nesse item.⁵
- ❺ Esse é um construtor alternativo – um uso comum para o decorador `@classmethod`.
- ❻ Se `obj` é um mapeamento, cria um `FrozenJSON` com ele.

⁵ É nessa linha que uma exceção `KeyError` pode ocorrer, na expressão `self.__data[name]`. Ela deve ser tratada, e uma `AttributeError` deve ser levantada em seu lugar, pois é isso que se espera de `__getattr__`. O leitor aplicado está convidado a implementar o tratamento de erro como exercício.

- ⑦ Se for uma `MutableSequence`, deve ser uma lista,⁶ portanto criamos uma `list` passando todos os itens de `obj` recursivamente para `.build()`.
- ⑧ Se não for um `dict` nem uma `list`, devolve o item como está.

Observe que não há caching nem transformação do feed original. À medida que o feed é percorrido, as estruturas de dados aninhadas são convertidas repetidamente em `FrozenJSON`. Mas não há problemas para um conjunto de dados desse tamanho e para um script que será usado apenas para explorar ou converter os dados.

Qualquer script que gere ou emule nomes de atributos dinâmicos a partir de uma fonte qualquer de dados deve lidar com um problema: as chaves nos dados originais podem não ser nomes válidos para atributos. A próxima seção trata esse assunto.

O problema do nome de atributo inválido

A classe `FrozenJSON` tem uma limitação: não há tratamento especial para nomes de atributo que sejam palavras reservadas em Python. Por exemplo, se você criar um objeto assim:

```
>>> grad = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```

Você não poderá ler `grad.class` porque `class` é uma palavra reservada em Python:

```
>>> grad.class
File "<stdin>", line 1
  grad.class
    ^
SyntaxError: invalid syntax
```

É claro que você sempre pode usar:

```
>>> getattr(grad, 'class')
1982
```

Mas a ideia de `FrozenJSON` é oferecer um acesso conveniente aos dados, portanto uma solução melhor é verificar se uma chave no mapeamento fornecido a `FrozenJSON`. `__init__` é uma palavra reservada e, se for, concatenar um `_` a ela de modo que o atributo possa ser lido assim:

```
>>> grad.class_
1982
```

Isso pode ser feito substituindo o código de uma linha de `__init__` do exemplo 19.5 pela versão do exemplo 19.6.

⁶ A fonte de dados é JSON, e os únicos tipos de coleção em dados JSON são `dict` e `list`.

Exemplo 19.6 – `explore1.py`: concatena um `_` aos nomes de atributo que são palavras reservadas em Python

```
def __init__(self, mapping):
    self.__data = {}
    for key, value in mapping.items():
        if keyword.iskeyword(key): ❶
            key += '_'
        self.__data[key] = value
```

❶ A função `keyword.iskeyword(...)` é exatamente o que precisamos; para usá-la, o módulo `keyword` precisa ser importado, o que não foi mostrado nesse trecho de código.

Um problema semelhante pode surgir se uma chave nos dados JSON não for um identificador válido em Python:

```
>>> x = FrozenJSON({'2be': 'or not'})
>>> x.2be
File "<stdin>", line 1
    x.2be
      ^
SyntaxError: invalid syntax
```

Chaves problemáticas como essas são fáceis de detectar em Python 3 porque a classe `str` oferece o método `s.isidentifier()`, que diz se `s` é um identificador válido em Python de acordo com a gramática da linguagem. Contudo transformar uma chave que não seja um identificador válido em um nome de atributo válido não é trivial. Duas soluções simples seriam levantar uma exceção ou substituir as chaves inválidas por nomes genéricos como `attr_0`, `attr_1`, e assim sucessivamente. Para simplificar, não me preocuparei com esse problema.

Após ter discutido um pouco os nomes de atributos dinâmicos, vamos nos voltar para outro recurso essencial de `FrozenJSON`: a lógica do método de classe `build`, usado por `__getattr__` para devolver um tipo diferente de objeto, de acordo com o valor do atributo acessado, de modo que estruturas aninhadas são convertidas em instâncias de `FrozenJSON` ou em listas de instâncias de `FrozenJSON`.

Em vez de um método de classe, a mesma lógica pode ser implementada com o método especial `__new__`, como veremos a seguir.

Criação flexível de objetos com `__new__`

Com frequência, nos referimos a `__init__` como o método construtor, mas isso é porque adotamos o jargão de outras linguagens. O método especial que realmente constrói uma instância é `__new__`: é um método de classe (mas tem tratamento especial,

portanto o decorador `@classmethod` não é usado) e deve devolver uma instância. Essa instância, por sua vez, será passada como o primeiro argumento `self` de `__init__`. Como `__init__` recebe uma instância quando é chamado e, na verdade, está proibido de devolver qualquer dado, `__init__` é realmente um “inicializador”. O verdadeiro construtor é `__new__` – que raramente precisamos implementar, pois a implementação herdada de `object` é suficiente.

O caminho que acabamos de descrever, de `__new__` a `__init__`, é o mais comum, mas não é o único. O método `__new__` também pode devolver uma instância de uma classe diferente e, quando isso acontece, o interpretador não chama `__init__`.

Em outras palavras, o processo de construir um objeto em Python pode ser sintetizado pelo pseudocódigo a seguir:

```
# pseudocódigo para construção de um objeto
def object_maker(the_class, some_arg):
    new_object = the_class.__new__(some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

# os comandos a seguir são, grosso modo, equivalentes
x = Foo('bar')
x = object_maker(Foo, 'bar')
```

O exemplo 19.7 mostra uma variação de `FrozenJSON`, em que a lógica do método de classe `build` anterior foi transferida para `__new__`.

Exemplo 19.7 – `explore2.py`: usando `new` em vez de `build` para construir objetos novos que podem ou não ser instâncias de `FrozenJSON`

```
from collections import abc
```

```
class FrozenJSON:
```

```
    """Uma fachada somente de leitura para navegar por um objeto JSON ou similar
       usando notação de atributo
    """
```

```
    def __new__(cls, arg): ❶
        if isinstance(arg, abc.Mapping):
            return super().__new__(cls) ❷
        elif isinstance(arg, abc.MutableSequence): ❸
            return [cls(item) for item in arg]
        else:
            return arg
```

```

def __init__(self, mapping):
    self.__data = {}
    for key, value in mapping.items():
        if iskeyword(key):
            key += '_'
        self.__data[key] = value

def __getattr__(self, name):
    if hasattr(self.__data, name):
        return getattr(self.__data, name)
    else:
        return FrozenJSON(self.__data[name]) ❹

```

- ❶ Como um método de classe, o primeiro argumento que `__new__` recebe é a própria classe, e os argumentos restantes são os mesmos recebidos por `__init__`, exceto `self`.
- ❷ O comportamento-padrão é delegar para o `__new__` de uma superclasse. Nesse caso, chamamos `__new__` da classe-base `object`, passando `FrozenJSON` como único argumento.
- ❸ As linhas restantes de `__new__` são exatamente iguais à linha do método `build` anterior.
- ❹ É nesse ponto que `FrozenJSON.build` era chamado antes; agora simplesmente chamamos o construtor `FrozenJSON`.

O método `__new__` recebe a classe como primeiro argumento porque, normalmente, o objeto criado será uma instância dessa classe. Assim, em `FrozenJSON.__new__`, quando a expressão `super().__new__(cls)` chama efetivamente `object.__new__(FrozenJSON)`, a instância criada pela classe `object`, na verdade, é uma instância de `FrozenJSON` – ou seja, o atributo `__class__` da nova instância armazenará uma referência a `FrozenJSON` –, apesar de a construção propriamente dita ser realizada por `object.__new__` implementado em C, nas entranhas do interpretador.

Há um defeito óbvio em como o feed JSON da OSCON está estruturado: o evento no índice 40, cujo título é 'There *Will* Be Bugs', tem dois palestrantes, 3471 e 5199, mas encontrá-los não é fácil, pois são números identificadores e a lista `Schedule.speakers` não está indexada por eles. O campo `venue`, presente em todo registro `event`, também armazena um número identificador, mas encontrar o registro `venue` correspondente exige uma busca linear na lista `Schedule.venues`. Nossa próxima tarefa é reestruturar os dados e então automatizar a recuperação dos registros associados.

Reestruturando o feed da OSCON com `shelve`

O nome curioso do módulo `shelve` faz sentido quando percebemos que `pickle` é o nome do formato de serialização de objetos Python – e do módulo que converte objetos de/para esse formato. Como vidros de pickles (pickles) são guardados em prateleiras

(shelves), é natural que a armazenagem de `pickle` seja feita por `shelve`, pois o verbo “to shelve” significa “guardar em uma prateleira”.

A função de alto nível `shelve.open` devolve uma instância de `shelve.Shelf` – um banco de dados do tipo chave-valor simples baseado no módulo `dbm`, com as seguintes características:

- `shelve.Shelf` herda de `abc.MutableMapping`, portanto oferece os métodos essenciais que esperamos de um mapeamento.
- Além disso, `shelve.Shelf` oferece outros métodos de gerenciamento de E/S, como `sync` e `close`; ele também é um gerenciador de contexto.
- Chaves e valores são salvos sempre que um novo valor é atribuído a uma chave.
- As chaves têm que ser strings.
- Os valores têm que ser objetos que o módulo `pickle` consiga serializar.

Consulte a documentação dos módulos `shelve` (<https://docs.python.org/3/library/shelve.html>), `dbm` (<https://docs.python.org/3/library/dbm.html>) e `pickle` (<https://docs.python.org/3/library/pickle.html>) para ver os detalhes e as ressalvas. O que importa para nós no momento é que `shelve` oferece um modo simples e eficiente de reorganizar os dados da agenda da OSCON: leremos todos os registros do arquivo JSON e os salvaremos em um `shelve.Shelf`. Cada chave será composta do tipo do registro e do número identificador (por exemplo, `'event.33950'` ou `'speaker.3471'`), e o valor será uma instância de uma nova classe `Record` que vamos apresentar em seguida.

O exemplo 19.8 mostra os doctests para o script `schedule1.py` que usa `shelve`. Para testá-lo interativamente, execute o script com `python -i schedule1.py` para obter um prompt no console com o módulo já carregado. A função `load_db` faz o trabalho pesado: chama `oscon.feed.load` (do exemplo 19.2) para ler os dados JSON e salva cada registro como uma instância de `Record` no objeto `Shelf` passado como `db`. Depois disso, recuperar um registro de palestrante é simples: basta usar `speaker = db['speaker.3471']`.

Exemplo 19.8 – Testando a funcionalidade oferecida por `schedule1.py` (Exemplo 19.9)

```
>>> import shelve
>>> db = shelve.open(DB_NAME) ❶
>>> if CONFERENCE not in db: ❷
...     load_db(db) ❸
...
>>> speaker = db['speaker.3471'] ❹
>>> type(speaker) ❺
```

```
<class 'schedule1.Record'>
>>> speaker.name, speaker.twitter ❹
('Anna Martelli Ravenscroft', 'annaraven')
>>> db.close() ❺
```

- ❶ `shelve.open` abre um arquivo de banco de dados existente ou um novo, criado nesse instante.
- ❷ Uma maneira rápida de determinar se o banco de dados está preenchido é procurar uma chave conhecida, nesse caso, `conference.115` – a chave do registro único `conference`.⁷
- ❸ Se o banco de dados estiver vazio, chama `load_db(db)` para carregá-lo.
- ❹ Busca um registro de `speaker`.
- ❺ É uma instância da classe `Record` definida no exemplo 19.9.
- ❻ Cada instância de `Record` implementa um conjunto customizado de atributos que reflete os campos do registro JSON subjacente.
- ❼ Nunca se esqueça de fechar um `shelve.Shelf`. Se possível, use um bloco `with` para garantir que o `Shelf` seja fechado.⁸

O código de *schedule1.py* está no exemplo 19.9.

Exemplo 19.9 – *schedule1.py*: para explorar os dados da agenda da OSCON salvos em um `shelve.Shelf`

```
import warnings
import osconfeed ❶

DB_NAME = 'data/schedule1_db'
CONFERENCE = 'conference.115'

class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs) ❷

def load_db(db):
    raw_data = osconfeed.load() ❸
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items(): ❹
        record_type = collection[:-1] ❺
```

7 Eu também poderia ter usado `len(db)`, mas isso seria custoso em um banco de dados `dbm` grande.

8 Um ponto fraco fundamental de `doctest` é a falta de suporte a `setup` e `teardown`, para assegurar a inicialização e a liberação de recursos. Por isso escrevi a maior parte dos testes de *schedule1.py* usando `py.test`, e você pode vê-los no exemplo A.12.

```

for record in rec_list:
    key = '{}.{}'.format(record_type, record['serial']) ❹
    record['serial'] = key ❺
    db[key] = Record(**record) ❻

```

- ❶ Carrega o módulo *osconfeed.py* do exemplo 19.2.
- ❷ Esse é um atalho comum para construir uma instância com atributos criados a partir de argumentos nomeados (explicações detalhadas a seguir).
- ❸ Esse código pode buscar o feed JSON da Web se não houver uma cópia local.
- ❹ Itera pelas coleções (por exemplo, 'conferences', 'events' etc.).
- ❺ `record_type` é definido com o nome da coleção sem o 's' final (ou seja, 'events' torna-se 'event').
- ❻ Constrói `key` a partir de `record_type` e do campo 'serial'.
- ❼ Atualiza o campo 'serial' com a chave (`key`) completa.
- ❽ Cria uma instância de `Record` e salva-a no banco de dados com `key`.

O método `Record.__init__` ilustra um hack popular em Python. Lembre-se que o `__dict__` de um objeto é onde seus atributos são mantidos – a menos que `__slots__` esteja declarado na classe, como vimos na seção “Economizando espaço com o atributo de classe `__slots__`” na página 307. Então atualizar o `__dict__` de uma instância com um mapeamento é um modo rápido de criar um punhado de atributos nessa instância.⁹



Não repetirei os detalhes que discutimos antes na seção “O problema do nome de atributo inválido” na página 651, mas, dependendo do contexto da aplicação, a classe `Record` talvez precise lidar com chaves que não sejam nomes válidos de atributos.

A definição de `Record` no exemplo 19.9 é tão simples que você pode estar se perguntando por que não a usamos antes, em vez de usar o mais complicado `FrozenJSON`. Há alguns motivos. Em primeiro lugar, `FrozenJSON` funciona convertendo recursivamente os mapeamentos e as listas aninhados; `Record` não precisa disso porque nosso conjunto de dados convertidos não tem mapeamentos aninhados em mapeamentos ou listas. Os registros contêm apenas strings, inteiros, listas de strings e listas de inteiros. Um segundo motivo é que `FrozenJSON` oferece acesso aos atributos do `dict` `__data` embutido – que usamos para chamar métodos como `keys` –, e agora não precisamos dessa funcionalidade.

⁹ A propósito, `Bunch` (que quer dizer um punhado, em inglês) é o nome da classe usada por Alex Martelli para compartilhar essa dica em uma receita de 2001 cujo título é “The simple but handy *collector of a bunch of named stuff* class” (A classe simples, porém conveniente, para reunir um punhado de itens nomeados, <http://bit.ly/1cPM8T3>).



A biblioteca-padrão de Python oferece pelo menos duas classes semelhantes ao nosso `Record`, em que cada instância tem um conjunto arbitrário de atributos criados a partir de argumentos nomeados recebidos pelo construtor: `multiprocessing.Namespace` (documentação em <http://bit.ly/1cPLZzd>; código-fonte em <http://bit.ly/1cPM2uJ>) e `argparse.Namespace` (documentação em <http://bit.ly/1cPM1qG>; código-fonte em <http://bit.ly/1cPM4Ti>). Implementei `Record` para enfatizar a essência da ideia: `__init__` atualizando o `__dict__` da instância.

Após reorganizar o conjunto de dados da agenda como fizemos, podemos agora estender a classe `Record` para oferecer um serviço útil: recuperar automaticamente registros `venue` e `speaker` referenciados em um registro `event`. É semelhante ao que o ORM de Django faz quando um campo `models.ForeignKey` é acessado: em vez da chave, você obtém o objeto do modelo relacionado. Usaremos propriedades para fazer isso no próximo exemplo.

Recuperação de registros relacionados usando propriedades

O objetivo desta próxima versão é: dado um registro `event` recuperado do shelf, ler seus atributos `venue` ou `speakers` não devolverá números identificadores, mas objetos de registro completos. Veja a interação parcial no exemplo 19.10.

Exemplo 19.10 – Extraído dos doctests de `schedule2.py`

```
>>> DbRecord.set_db(db) ❶
>>> event = DbRecord.fetch('event.33950') ❷
>>> event ❸
<Event 'There *Will* Be Bugs'>
>>> event.venue ❹
<DbRecord serial='venue.1449'>
>>> event.venue.name ❺
'Portland 251'
>>> for spkr in event.speakers: ❻
...     print('{0.serial}: {0.name}'.format(spkr))
...
speaker.3471: Anna Martelli Ravenscroft
speaker.5199: Alex Martelli
```

- ❶ `DbRecord` estende `Record`, acrescentando suporte a banco de dados: para funcionar, `DbRecord` precisa ser configurado com uma referência a um banco de dados.
- ❷ O método de classe `DbRecord.fetch` recupera registros de qualquer tipo.
- ❸ Observe que `event` é uma instância da classe `Event`, que estende `DbRecord`.
- ❹ Acessar `event.venue` devolve uma instância de `DbRecord`.

- ➊ Agora é fácil descobrir o nome de um `event.venue`. Essa desreferenciação automática é o objetivo desse exemplo.
- ➋ Também podemos iterar pela lista `event.speakers`, recuperando `DbRecords` que representam cada palestrante.

A figura 19.1 apresenta uma visão geral das classes que estudaremos nesta seção:

Record

O método `__init__` é o mesmo de *schedule1.py* (Exemplo 19.9); o método `__eq__` foi adicionado para facilitar os testes.

DbRecord

Subclasse de `Record` adicionando um atributo de classe `__db`, métodos estáticos `set_db` e `get_db` para escrever/ler esse atributo, um método de classe `fetch` para recuperar registros do banco de dados e um método de instância `__repr__` para suporte à depuração e testes.

Event

Subclasse de `DbRecord` adicionando propriedades `venue` e `speakers` para recuperar registros associados e um método `__repr__` especializado.

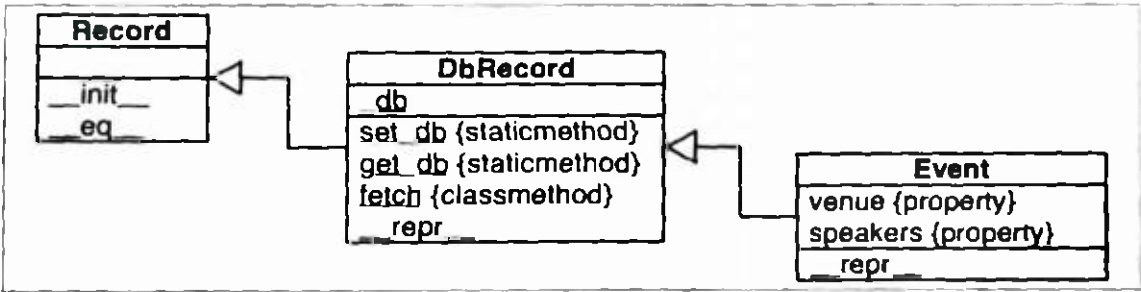


Figura 19.1 – Diagrama de classes UML para uma classe `Record` melhorada e duas subclasses: `DbRecord` e `Event`.

O atributo de classe `DbRecord.__db` armazena uma referência ao banco de dados `shelve.Shelf` aberto, para que possa ser usado pelo método `DbRecord.fetch` e pelas propriedades `Event.venue` e `Event.speakers` que dependem dele. Implementei `__db` como um atributo de classe privado, com métodos getter e setter convencionais, porque queria protegê-lo de sobrescritas acidentais. Não usei uma propriedade para administrar `__db` devido a um fato crucial: propriedades são atributos de classe projetados para gerenciar atributos de instância.¹⁰

¹⁰ O tópico “Class-level read only properties in Python” (Propriedades somente de leitura no nível de classes em Python, <http://bit.ly/1cPMnNZ>) em StackOverflow tem soluções para atributos somente de leitura em classes, incluindo uma de Alex Martelli. As soluções exigem metaclasses, então talvez você queira ler o capítulo 21 antes de estudá-las.

O código desta seção está no módulo *schedule2.py* no repositório de código de *Python fluente* (<https://github.com/fluentpython/example-code>). Como o módulo tem mais de cem linhas, ele será apresentado em partes.¹¹

Os primeiros comandos de *schedule2.py* estão no exemplo 19.11.

Exemplo 19.11 – *schedule2.py*: importações, constantes e a classe *Record* melhorada

```
import warnings
import inspect ❶

import osconfeed

DB_NAME = 'data/schedule2_db' ❷
CONFERENCE = 'conference.115'

class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __eq__(self, other): ❸
        if isinstance(other, Record):
            return self.__dict__ == other.__dict__
        else:
            return NotImplemented
```

- ❶ `inspect` será usado na função `load_db` (Exemplo 19.14).
- ❷ Como estamos armazenando instâncias de classes diferentes, criamos e usamos um arquivo diferente de banco de dados, `'schedule2_db'`, em vez de usar `'schedule_db'` do exemplo 19.9.
- ❸ Um método `__eq__` é sempre conveniente para testes.



Em Python 2, somente classes “new style” (estilo novo) suportam propriedades. Para escrever uma classe new-style em Python 2, você precisa herdar direta ou indiretamente de `object`. A classe `Record` no exemplo 19.11 é a classe-base de uma hierarquia que usará propriedades, portanto em Python 2 a declaração dessa classe deve começar com:^(*)

```
class Record(object):
    # etc...
```

- (*) Herdar explicitamente de `object` em Python 3 não está errado; é apenas redundante porque todas as classes agora são “new-style classes”. Esse é um exemplo em que romper com o passado deixou a linguagem mais limpa. Se o mesmo código precisar executar em Python 2 e em Python 3, deve-se herdar explicitamente de `object`.

¹¹ A listagem completa de *schedule2.py* está no exemplo A.13, juntamente com os scripts *py.test* na seção “Capítulo 19: scripts e testes para agenda da OSCON” na página 775.

As próximas classes definidas em *schedule2.py* são um tipo customizado de exceção e *DbRecord*. Veja o exemplo 19.12.

Exemplo 19.12 – *schedule2.py*: as classes *MissingDatabaseError* e *DbRecord*

```
class MissingDatabaseError(RuntimeError):
    """Levantada quando um banco de dados é necessário, mas não foi definido.""" ❶

class DbRecord(Record): ❷
    __db = None ❸
    @staticmethod ❹
    def set_db(db):
        DbRecord.__db = db ❺
    @staticmethod ❻
    def get_db():
        return DbRecord.__db
    @classmethod ❼
    def fetch(cls, ident):
        db = cls.get_db()
        try:
            return db[ident] ❽
        except TypeError:
            if db is None: ❹
                msg = "database not set; call '{}.set_db(my_db)'"
                raise MissingDatabaseError(msg.format(cls.__name__))
            else: ❿
                raise
    def __repr__(self):
        if hasattr(self, 'serial'): ⓫
            cls_name = self.__class__.__name__
            return '<{} serial={!r}>'.format(cls_name, self.serial)
        else:
            return super().__repr__() ⓬
```

- ❶ Exceções customizadas podem ser classes marcadoras, sem corpo. Uma docstring explicando o uso da exceção é melhor que um mero comando `pass`.
- ❷ *DbRecord* estende *Record*.
- ❸ O atributo de classe `__db` armazenará uma referência ao banco de dados *shelve.Shelf* que foi aberto.

- ❹ `set_db` é um `staticmethod` para deixar explícito que seu efeito é sempre exatamente o mesmo, independentemente de como é chamado.
- ❺ Mesmo que esse método seja chamado como `Event.set_db(my_db)`, o atributo `__db` será definido na classe `DbRecord`.
- ❻ `get_db` também é um `staticmethod` porque sempre devolverá o objeto referenciado por `DbRecord.__db`, independentemente de como for chamado.
- ❼ `fetch` é um método de classe, portanto seu comportamento é mais fácil de customizar em subclasses.
- ❽ Recupera o registro com a chave `ident` do banco de dados.
- ❾ Se tivermos um `TypeError` e `db` for `None`, levantará uma exceção customizada explicando que o banco de dados deve estar definido.
- ❿ Caso contrário, levanta a exceção novamente porque não sabemos como tratá-la.
- ⓫ Se o registro tem um atributo `serial`, usa-o na representação em string.
- ⓬ Caso contrário, usa o `__repr__` herdado como default.

Agora chegamos à parte mais importante do exemplo: a classe `Event`, listada no exemplo 19.13.

Exemplo 19.13 – `schedule2.py`: a classe `Event`

```
class Event(DbRecord): ❶
    @property
    def venue(self):
        key = 'venue.{}'.format(self.venue_serial)
        return self.__class__.fetch(key) ❷

    @property
    def speakers(self):
        if not hasattr(self, '_speaker_objs'): ❸
            spkr_serials = self.__dict__['speakers'] ❹
            fetch = self.__class__.fetch ❺
            self._speaker_objs = [fetch('speaker.{}'.format(key))
                                  for key in spkr_serials] ❻
        return self._speaker_objs ❼

    def __repr__(self):
        if hasattr(self, 'name'): ❽
            cls_name = self.__class__.__name__
            return '<{} {}>'.format(cls_name, self.name)
```

```
else:  
    return super().__repr__() ❶
```

- ❶ Event estende DbRecord.
- ❷ A propriedade `venue` constrói uma `key` a partir do atributo `venue_serial` e passa-a para o método de classe `fetch`, herdado de `DbRecord` (veja a explicação após esse exemplo).
- ❸ A propriedade `speakers` verifica se o registro tem um atributo `_speaker_objs`.
- ❹ Se não tiver, o atributo `'speakers'` será recuperado diretamente do `__dict__` da instância para evitar uma recursão infinita, pois o nome público dessa propriedade também é `speakers`.
- ❺ Obtém uma referência ao método de classe `fetch` (a justificativa para isso será explicada em breve).
- ❻ `self._speaker_objs` é carregado com uma lista de registros `speaker` usando `fetch`.
- ❼ Essa lista é devolvida.
- ❽ Se o registro tem um atributo `name`, usa-o na representação em string.
- ❾ Caso contrário, usa o `__repr__` herdado como default.

Na propriedade `venue` do exemplo 19.13, a última linha devolve `self.__class__.fetch(key)`. Por que não escrever isso simplesmente como `self.fetch(key)`? A fórmula mais simples funciona com o conjunto de dados específico do feed da OSCON porque não há nenhum registro de eventos com uma chave `'fetch'`. Se um único registro de eventos tivesse uma chave chamada `'fetch'`, então, nessa instância específica de `Event`, a referência `self.fetch` recuperaria o valor desse campo, em vez de recuperar o método de classe `fetch` que `Event` herda de `DbRecord`. É um bug sutil, e poderia passar facilmente pelos testes e explodir somente no ambiente de produção, quando os registros de local (`venue`) ou de palestrante (`speaker`) associados a esse registro `Event` específico fossem recuperados.



Ao criar nomes de atributos de instância a partir de dados, há sempre o risco de haver bugs devido ao encobrimento de atributos da classe (por exemplo, métodos) ou à perda de dados resultante de sobrescritas acidentais de atributos de instância existentes. Essa ressalva provavelmente é o principal motivo pelo qual, por padrão, os dicionários Python não são como objetos de JavaScript, afinal.

Se a classe `Record` se comportasse mais como um mapeamento, implementando um `__getitem__` dinâmico em vez de um `__getattr__` dinâmico, não haveria risco de bugs resultantes de sobrescrita ou encobrimento. Um mapeamento especializado provavelmente é a maneira pythônica de implementar `Record`. Mas, se eu optasse por esse caminho, não estaríamos refletindo sobre os truques e as armadilhas da programação de atributos dinâmicos.

A última parte desse exemplo é a função `load_db` revisada no exemplo 19.14.

Exemplo 19.14 – `schedule2.py`: a função `load_db`

```
def load_db(db):
    raw_data = osconfeed.load()
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():
        record_type = collection[:-1] ❶
        cls_name = record_type.capitalize() ❷
        cls = globals().get(cls_name, DbRecord) ❸
        if inspect.isclass(cls) and issubclass(cls, DbRecord): ❹
            factory = cls ❺
        else:
            factory = DbRecord ❻
        for record in rec_list: ❼
            key = '{}.{}'.format(record_type, record['serial'])
            record['serial'] = key
            db[key] = factory(**record) ❽
```

- ❶ Até agora, nenhuma alteração em relação ao `load_db` em *schedule1.py* (Exemplo 19.9).
- ❷ Converte as iniciais de `record_type` para letra maiúscula a fim de obter um nome de classe em potencial (por exemplo, 'event' torna-se 'Event').
- ❸ Obtém um objeto por esse nome a partir do escopo global do módulo; obtém `DbRecord` se esse objeto não existir.
- ❹ Se o objeto que acabou de ser recuperado é uma classe, e é subclasse de `DbRecord`...
- ❺ ... vincula o nome `factory` a ele. Isso quer dizer que `factory` pode ser qualquer subclasse de `DbRecord`, dependendo do `record_type`.
- ❻ Caso contrário, vincula o nome `factory` a `DbRecord`.
- ❼ O laço `for` que cria a chave (`key`) e salva os registros é o mesmo de antes, exceto que...
- ❽ ... o objeto armazenado no banco de dados é construído por `factory`, que pode ser `DbRecord` ou uma subclasse selecionada de acordo com `record_type`.

Observe que o único `record_type` que tem uma classe especial é `Event`, mas se classes chamadas `Speaker` ou `Venue` forem implementadas, `load_db` usará essas classes automaticamente quando criar e salvar registros, em vez de usar a classe `DbRecord` default.

Até agora, os exemplos deste capítulo foram projetados para mostrar várias técnicas de implementação de atributos dinâmicos usando ferramentas básicas como `__getattr__`, `hasattr`, `getattr`, `@property` e `__dict__`.

Propriedades muitas vezes são usadas para impor regras de negócio, mudando um atributo público que passa a ser um atributo gerenciado por um getter e um setter sem afetar o código do cliente, como mostra a próxima seção.

Usando uma propriedade para validação de atributo

Até agora, vimos apenas o decorador `@property` usado para implementar propriedades somente de leitura. Nesta seção, criaremos uma propriedade para leitura/escrita.

LineItem tomada #1: classe para um item de um pedido

Pense em um aplicativo para uma loja que vende alimentos orgânicos a granel, onde os clientes possam comprar castanhas, frutas secas ou cereais por peso. Nesse sistema, cada pedido seria uma sequência de itens, e cada item poderia ser representado por uma classe, como mostra o exemplo 19.15.

Exemplo 19.15 – `bulkfood_v1.py`: a classe `LineItem` mais simples

```
class LineItem:
    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price
    def subtotal(self):
        return self.weight * self.price
```

É bonito e simples. Talvez simples demais. O exemplo 19.16 mostra um problema.

Exemplo 19.16 – Um peso negativo resulta em um subtotal negativo

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> raisins.subtotal()
69.5
>>> raisins.weight = -20 # entra lixo...
>>> raisins.subtotal() # sai lixo...
-139.0
```

É um exemplo fictício, mas não tão fora da realidade como você possa estar imaginando. Veja uma história real do início da Amazon.com:

Descobrimos que os clientes podiam pedir uma quantidade negativa de livros! E lançávamos um crédito em seus cartões de crédito com o valor e, suponho, esperávamos que eles nos enviassem os livros.¹²

— Jeff Bezos
Fundador e CEO da Amazon.com

Como corrigir esse problema? Poderíamos mudar a interface de `LineItem` de modo a usar um getter e um setter para o atributo `weight`. Esse seria o modo Java, e não seria errado.

Por outro lado, é natural ser capaz de definir o peso (`weight`) de um item simplesmente fazendo uma atribuição; talvez o sistema já esteja em ambiente de produção, com outras partes já acessando `item.weight` diretamente. Nesse caso, o modo Python seria substituir o atributo de dado por uma propriedade.

LineItem tomada #2: uma propriedade com validação

Implementar uma propriedade nos permitirá usar um getter e um setter, mas a interface de `LineItem` não mudará (ou seja, definir `weight` em um `LineItem` continuará sendo escrito como `raisins.weight = 12`).

O exemplo 19.17 lista o código de uma propriedade `weight` para leitura/escrita.

Exemplo 19.17 – `bulkfood_v2.py`: um `LineItem` com uma propriedade `weight`

class `LineItem`:

```
def __init__(self, description, weight, price):
    self.description = description
    self.weight = weight ❶
    self.price = price

def subtotal(self):
    return self.weight * self.price

@property ❷
def weight(self): ❸
    return self.__weight ❹

@weight.setter ❺
def weight(self, value):
    if value > 0:
        self.__weight = value ❻
    else:
        raise ValueError('value must be > 0') ❼
```

¹² Citação direta de Jeff Bezos na matéria “Birth of a Salesman” (Nascimento de um vendedor, <http://on.wsj.com/1EC18DI>) no *Wall Street Journal* (15 de outubro de 2011).

- ❶ Aqui o método `setter` da propriedade já está em uso, garantindo que nenhuma instância com `weight` negativo possa ser criada.
- ❷ `@property` decora o método `getter`.
- ❸ Os métodos que implementam uma propriedade têm o nome do atributo público: `weight`.
- ❹ O valor propriamente dito é armazenado em um atributo privado `__weight`.
- ❺ O `getter` decorado tem um atributo `.setter`, que também é um decorador; isso vincula o `getter` e o `setter`.
- ❻ Se o valor é maior que zero, escrevemos no `__weight` privado.
- ❼ Caso contrário, `ValueError` é levantada.

Observe como um `LineItem` com um peso inválido não pode mais ser criado:

```
>>> walnuts = LineItem('walnuts', 0, 10.00)
Traceback (most recent call last):
...
ValueError: value must be > 0
```

Agora protegemos `weight` de usuários que forneçam valores negativos. Embora os consumidores normalmente não possam definir o preço de um item, um erro não intencional ou um bug podem criar um `LineItem` com um `price` negativo. Para evitar isso, poderíamos também transformar `price` em uma propriedade, mas isso implicaria certa repetição em nosso código.

Lembre-se da citação de Paul Graham no capítulo 14: “Quando vejo padrões em meus programas, considero isso sinal de problema.” A solução para repetição é a abstração. Há duas maneiras de abstrair definições de propriedades: usando uma fábrica (factory) de propriedades ou uma classe descritora. A abordagem de classe descritora é mais flexível, e dedicaremos o capítulo 20 para discuti-la de forma completa. Propriedades, na verdade, são implementadas como classes descritoras. Porém aqui continuaremos a explorar as propriedades implementando uma fábrica de propriedades como uma função.

Mas, antes de podermos implementar uma fábrica de propriedades, devemos ter uma compreensão mais profunda das propriedades.

Uma visão apropriada das propriedades

Embora seja frequentemente usada como decorador, `property`, na verdade, é uma classe embutida. Em Python, funções e classes muitas vezes são intercambiáveis.

pois ambas são invocáveis e não há um operador `new` para instanciação de objetos, portanto invocar um construtor não é diferente de invocar uma função de fábrica. Ambas podem ser usadas como decoradores, desde que devolvam um novo invocável que seja um substituto adequado à função decorada.

Veja a assinatura completa do construtor `property`:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Todos os argumentos são opcionais e, se uma função não for especificada para um deles, a operação correspondente não será permitida pelo objeto de propriedade resultante.

O tipo `property` foi acrescentado em Python 2.2, mas a sintaxe de decorador `@` surgiu somente em Python 2.4; assim, por alguns anos, as propriedades eram definidas passando as funções de acesso como os dois primeiros argumentos.

A sintaxe “clássica” para definir propriedades sem decoradores está no exemplo 19.18.

Exemplo 19.18 – `bulkfood_v2b.py`: igual ao exemplo 19.17, mas sem usar decoradores

```
class LineItem:
```

```
    def __init__(self, description, weight, price):
```

```
        self.description = description
```

```
        self.weight = weight
```

```
        self.price = price
```

```
    def subtotal(self):
```

```
        return self.weight * self.price
```

```
    def get_weight(self): ❶
```

```
        return self.__weight
```

```
    def set_weight(self, value): ❷
```

```
        if value > 0:
```

```
            self.__weight = value
```

```
        else:
```

```
            raise ValueError('value must be > 0')
```

```
    weight = property(get_weight, set_weight) ❸
```

❶ Um getter comum.

❷ Um setter comum.

❸ Cria a `property` e vincula-a a um atributo público da classe.

A forma clássica é melhor que a sintaxe com decorador em algumas situações; o código da fábrica de propriedades que discutiremos em breve é um exemplo. Por outro

lado, em um corpo de classe com muitos métodos, os decoradores deixam explícitos quais são getters e setters, sem depender da convenção de usar prefixos `get` e `set` em seus nomes.

A presença de uma propriedade em uma classe afeta como atributos em instâncias dessa classe são encontrados de uma forma que, à primeira vista, pode ser surpreendente. A próxima seção explica isso.

Propriedades encobrem atributos de instância

Propriedades são sempre atributos da classe, mas elas, na verdade, gerenciam o acesso a atributos nas instâncias da classe.

Na seção “Sobrescrita de atributos de classe” na página 310, vimos que, quando uma instância e sua classe têm ambas um atributo de dado de mesmo nome, o atributo da instância sobrescreve, ou encobre, o atributo da classe – pelo menos quando acessamos por meio dessa instância. O exemplo 19.19 ilustra essa questão.

Exemplo 19.19 – Atributo de instância sobrescreve um atributo de dado da classe

```
>>> class Class: # ❶
...     data = 'the class data attr'
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>> obj = Class()
>>> vars(obj) # ❷
{}
>>> obj.data # ❸
'the class data attr'
>>> obj.data = 'bar' # ❹
>>> vars(obj) # ❺
{'data': 'bar'}
>>> obj.data # ❻
'bar'
>>> Class.data # ❼
'the class data attr'
```

- ❶ Define `Class` com dois atributos de classe: o atributo de dado `data` e a propriedade `prop`.
- ❷ `vars` devolve o `__dict__` de `obj`, mostrando que ele não tem atributos de instância.

- ❸ Ler de `obj.data` recupera o valor de `Class.data`.
- ❹ Escrever em `obj.data` cria um atributo de instância.
- ❺ Inspecciona a instância para ver o atributo da instância.
- ❻ Agora a leitura de `obj.data` recupera o valor do atributo da instância. Ao ler da instância `obj`, o `data` da instância encobre o `data` da classe.
- ❼ O atributo `Class.data` permanece intacto.

Vamos agora tentar sobrescrever o atributo `prop` na instância `obj`. Retomando a sessão de console anterior, temos o exemplo 19.20.

Exemplo 19.20 – Atributo de instância não encobre a propriedade da classe (continuação do exemplo 19.19)

```
>>> Class.prop # ❶
<property object at 0x1072b7408>
>>> obj.prop # ❷
'the prop value'
>>> obj.prop = 'foo' # ❸
Traceback (most recent call last):
...
AttributeError: can't set attribute
>>> obj.__dict__['prop'] = 'foo' # ❹
>>> vars(obj) # ❺
{'prop': 'foo', 'attr': 'bar'}
>>> obj.prop # ❻
'the prop value'
>>> Class.prop = 'baz' # ❼
>>> obj.prop # ❽
'foo'
```

- ❶ Ler `prop` diretamente de `Class` recupera o próprio objeto da propriedade sem executar seu método `getter`.
- ❷ Ler `obj.prop` executa o `getter` da propriedade.
- ❸ Tentar definir um atributo `prop` na instância falha.
- ❹ Colocar `'prop'` diretamente em `obj.__dict__` funciona.
- ❺ Podemos ver que `obj` agora tem dois atributos de instância: `attr` e `prop`.
- ❻ No entanto ler `obj.prop` continua executando o `getter` da propriedade. A propriedade não é encoberta por um atributo de instância.

- 7 Atribuir novo valor a `Class.prop` destrói o objeto da propriedade.
- 8 Agora `obj.prop` recupera o atributo da instância. `Class.prop` não é mais uma propriedade, portanto não encobre `obj.prop`.

Como demonstração final, acrescentaremos uma nova propriedade a `Class` e a veremos encobrir um atributo de instância. O exemplo 19.21 continua do ponto em que o exemplo 19.20 parou.

Exemplo 19.21 – Nova propriedade da classe encobre o atributo de instância existente (continuação do exemplo 19.20)

```
>>> obj.data # ❶
'bar'
>>> Class.data # ❷
'the class data attr'
>>> Class.data = property(lambda self: 'the "data" prop value') # ❸
>>> obj.data # ❹
'the "data" prop value'
>>> del Class.data # ❺
>>> obj.data # ❻
'bar'
```

- ❶ `obj.data` recupera o atributo `data` da instância.
- ❷ `Class.data` recupera o atributo `data` da classe.
- ❸ Sobrescreve `Class.data` com uma nova propriedade.
- ❹ `obj.data` agora está encoberto pela propriedade `Class.data`.
- ❺ Apaga a propriedade.
- ❻ `obj.data` agora lê o atributo `data` da instância novamente.

O ponto principal nesta seção é que uma expressão como `obj.attr` não procura `attr` começando por `obj`. A busca, na verdade, começa em `obj.__class__`, e somente se não houver uma propriedade chamada `attr` na classe é que Python procurará na instância `obj`. Essa regra se aplica não só a propriedades, mas a toda uma categoria de descritores, os *descritores dominantes* (overriding descriptors). Uma explicação mais detalhada dos descritores terá de esperar até o capítulo 20, em que veremos que propriedades, na verdade, são descritores dominantes.

Vamos agora voltar às propriedades. Toda unidade de código Python – módulos, funções, classes, métodos – pode ter uma docstring. O próximo assunto é como associar documentação às propriedades.

Documentação de propriedades

Quando ferramentas IDEs ou a função de console `help()` precisam exibir a documentação de uma propriedade, elas extraem as informações do atributo `__doc__` da propriedade.

Se usado com a sintaxe clássica de chamada, `property` recebe a string de documentação como o argumento `doc`:

```
weight = property(get_weight, set_weight, doc='weight in kilograms')
```

Quando `property` é usado como decorador, a docstring do método getter – aquele com o decorador `@property` – é usada como documentação da propriedade como um todo. A figura 19.2 mostra telas de ajuda geradas com o código do exemplo 19.22.

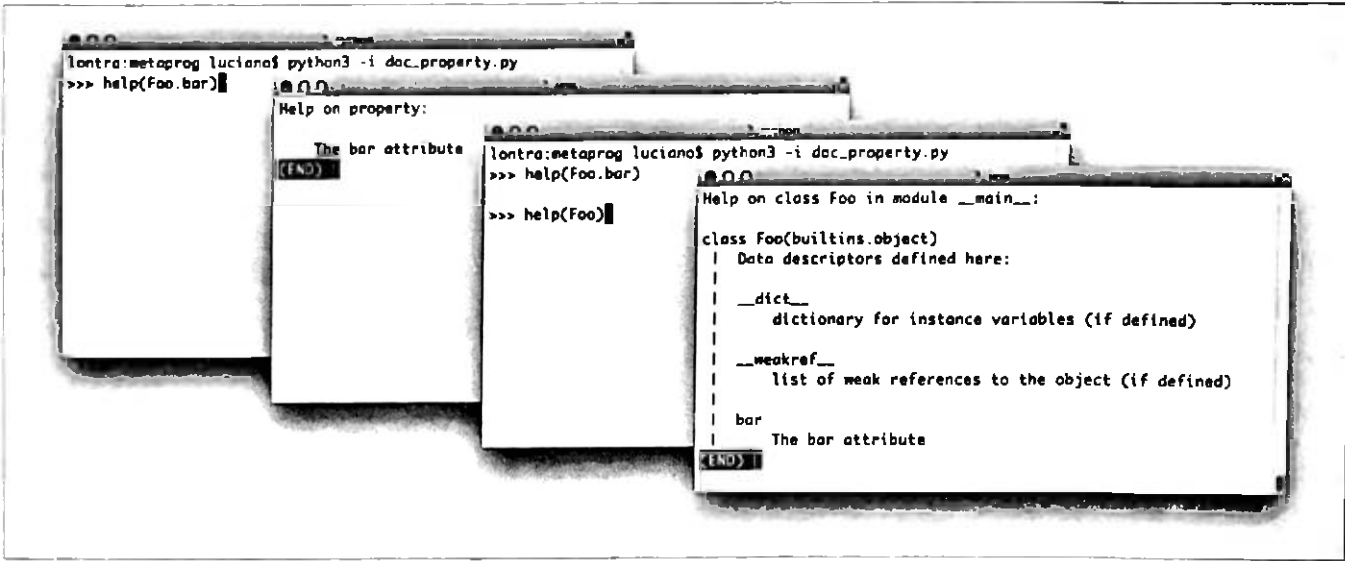


Figura 19.2 – Capturas de tela do console de Python quando os comandos `help(Foo.bar)` e `help(Foo)` são executados. Código-fonte no exemplo 19.22.

Exemplo 19.22 – Documentação de uma propriedade

```
class Foo:
    @property
    def bar(self):
        '''The bar attribute'''
        return self.__dict__['bar']

    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

Agora que já discutimos alguns aspectos essenciais das propriedades, vamos voltar à questão de proteger os atributos `weight` e `price` de `LineItem` para que só aceitem valores maiores que zero – mas sem implementar manualmente dois pares praticamente idênticos de getters/setters.

Implementando uma fábrica de propriedades

Criaremos uma fábrica de propriedades `quantity` – que recebeu esse nome porque os atributos gerenciados representam quantidades que não podem ser negativas nem iguais a zero na aplicação. O exemplo 19.23 mostra a aparência limpa da classe `LineItem` usando duas instâncias de propriedades `quantity`: uma para gerenciar o atributo `weight` e outra para `price`.

Exemplo 19.23 – `bulkfood_v2prop.py`: a fábrica de propriedade `quantity` em uso

```
class LineItem:
    weight = quantity('weight') ❶
    price = quantity('price') ❷

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight ❸
        self.price = price

    def subtotal(self):
        return self.weight * self.price ❹
```

- ❶ Usa a fábrica para definir a primeira propriedade customizada, `weight`, como um atributo da classe.
- ❷ Essa segunda chamada cria outra propriedade customizada, `price`.
- ❸ Aqui a propriedade já está ativa, garantindo que um valor de `weight` negativo ou zero será rejeitado.
- ❹ As propriedades também estão em uso aqui, recuperando os valores armazenados na instância.

Lembre-se de que propriedades são atributos da classe. Ao criar cada propriedade `quantity`, precisamos passar o nome do atributo de `LineItem` que será gerenciado por essa propriedade específica. Infelizmente é preciso digitar a palavra `weight` duas vezes nesta linha:

```
weight = quantity('weight')
```

Porém evitar essa repetição é complicado, pois a propriedade não tem como saber a que nome de atributo da classe ela será vinculada. Lembre-se: o lado direito de uma atribuição é avaliado antes, portanto, quando `quantity()` é chamada, o atributo de classe `price` nem sequer existe ainda.



Melhorar a propriedade `quantity` para que o usuário não precise digitar novamente o nome do atributo é um problema não trivial de metaprogramação. Veremos uma opção para contornar esse problema no capítulo 20, mas soluções melhores terão de esperar até o capítulo 21, pois exigem um decorador de classe ou uma metaclasses.

O exemplo 19.24 mostra a implementação da fábrica de propriedades `quantity`.¹³

Exemplo 19.24 – `bulkfood_v2prop.py`: a fábrica de propriedades `quantity`

```
def quantity(storage_name): ❶
    def qty_getter(instance): ❷
        return instance.__dict__[storage_name] ❸
    def qty_setter(instance, value): ❹
        if value > 0:
            instance.__dict__[storage_name] = value ❺
        else:
            raise ValueError('value must be > 0')
    return property(qty_getter, qty_setter) ❻
```

- ❶ O argumento `storage_name` determina o local em que os dados de cada propriedade são armazenados; para `weight`, o nome do local de armazenagem será `'weight'`.
- ❷ O primeiro argumento de `qty_getter` poderia se chamar `self`, mas isso seria estranho porque esse não é o corpo de uma classe; `instance` refere-se à instância de `LineItem` em que o atributo será armazenado.
- ❸ `qty_getter` referencia `storage_name`, portanto ela será preservada na closure dessa função; o valor é recuperado diretamente de `instance.__dict__` para passar por cima da propriedade e evitar uma recursão infinita.
- ❹ `qty_setter` é definido, também recebendo `instance` como primeiro argumento.
- ❺ `value` é armazenado diretamente em `instance.__dict__`, novamente passando por cima da propriedade.
- ❻ Cria um objeto customizado de propriedade e devolve-o.

As partes do exemplo 19.24 que merecem um estudo cuidadoso estão em torno da variável `storage_name`. Ao implementar cada propriedade da maneira tradicional, o nome do atributo em que você armazenará um valor é fixo nos métodos `getter` e `setter`. Mas aqui as funções `qty_getter` e `qty_setter` são genéricas e dependem da variável `storage_name` para saber em que local devem ler/escrever o atributo gerenciado

¹³ Esse código foi adaptado da “Recipe 9.21. Avoiding Repetitive Property Methods” (Evitar métodos repetitivos de propriedades) de *Python Cookbook*, 3E de David Beazley e Brian K. Jones (O’Reilly, <http://shop.oreilly.com/product/0636920027072.do>).

no `__dict__` da instância. Sempre que a fábrica `quantity` é chamada para criar uma propriedade, `storage_name` deve ser definido com um valor único.

As funções `qty_getter` e `qty_setter` serão encapsuladas pelo objeto `property` criado na última linha da função de fábrica. Posteriormente, quando forem chamadas para realizar suas obrigações, essas funções lerão o `storage_name` de suas closures para determinar o local em que devem recuperar/armazenar os valores dos atributos gerenciados.

No exemplo 19.25, criei e inspecionei uma instância de `LineItem`, expondo os atributos de armazenagem.

Exemplo 19.25 – `bulkfood_v2prop.py`: a fábrica de propriedades `quantity`

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price ❶
(8, 13.95)
>>> sorted(vars(nutmeg).items()) ❷
[('description', 'Moluccan nutmeg'), ('price', 13.95), ('weight', 8)]
```

- ❶ Lendo `weight` e `price` por meio das propriedades encobrindo os atributos de instância de mesmo nome.
- ❷ Usando `vars` para inspecionar a instância `nutmeg`: vemos aqui os atributos da instância usados para armazenar os valores.

Observe como as propriedades criadas pela nossa fábrica aproveitam o comportamento descrito na seção “Propriedades encobrem atributos de instância” na página 669: a propriedade `weight` encobre o atributo de instância `weight` de modo que toda referência a `self.weight` ou `nutmeg.weight` é tratada pelas funções da propriedade, e a única maneira de passar por cima do mecanismo da propriedade é acessando diretamente o `__dict__` da instância.

O código do exemplo 19.25 pode ser um pouco complicado, mas é conciso: tem o mesmo tamanho do par `getter/setter` decorado que define apenas a propriedade `weight` no exemplo 19.17. A definição de `LineItem` no exemplo 19.23 fica muito melhor sem `getter/setters` para atrapalhar.

Em um sistema de verdade, esse mesmo tipo de validação pode aparecer em muitos campos, em várias classes, e a `factory quantity` seria colocada em um módulo utilitário a ser reutilizado repetidamente. Em algum momento, essa fábrica simples poderia ser refatorada como uma classe descritora mais extensível, com subclasses especializadas executando validações diferentes. Faremos isso no capítulo 20.

Vamos agora encerrar a discussão sobre propriedades com a questão da remoção de atributos.

Tratando a remoção de atributos

Lembre-se de que, de acordo com o tutorial de Python, atributos de objetos podem ser apagados com o comando `del`:

```
del my_object.an_attribute
```

Na prática, apagar atributos não é algo que fazemos no dia a dia em Python, e a necessidade de tratar isso com uma propriedade é mais incomum ainda. Contudo, é suportado, e consigo pensar em um exemplo tolo para demonstrar isso.

Em uma definição de propriedade, o decorador `@my_property.deleter` é usado para encapsular o método responsável por apagar o atributo gerenciado pela propriedade. Como prometido, o exemplo 19.26 é um exemplo tolo que mostra como implementar a remoção de uma propriedade.

Exemplo 19.26 – `blackknight.py`: inspirado no personagem Cavaleiro Negro (Black Knight) de “Monty Python: em busca do cálice sagrado”

```
class BlackKnight:
    def __init__(self):
        self.members = ['an arm', 'another arm',
                        'a leg', 'another leg']
        self.phrases = ["'Tis but a scratch.",
                        "It's just a flesh wound.",
                        "I'm invincible!",
                        "All right, we'll call it a draw."]

    @property
    def member(self):
        print('next member is:')
        return self.members[0]

    @member.deleter
    def member(self):
        text = 'BLACK KNIGHT (loses {})\n-- {}'
        print(text.format(self.members.pop(0), self.phrases.pop(0)))
```

Os doctests em `blackknight.py` estão no exemplo 19.27.

Exemplo 19.27 – `blackknight.py`: doctests para o exemplo 19.26 (o Cavaleiro Negro jamais aceita a derrota)

```
>>> knight = BlackKnight()
>>> knight.member
next member is:
'an arm'
```



```
>>> del knight.member
BLACK KNIGHT (loses an arm)
.. 'Tis but a scratch.
>>> del knight.member
BLACK KNIGHT (loses another arm)
.. It's just a flesh wound.
>>> del knight.member
BLACK KNIGHT (loses a leg)
.. I'm invincible!
>>> del knight.member
BLACK KNIGHT (loses another leg)
.. All right, we'll call it a draw.
```

Usando a sintaxe clássica de chamada em vez de decoradores, o argumento `fdel` é utilizado para definir a função para apagar atributos. Por exemplo, a propriedade `member` seria implementada assim no corpo da classe `BlackKnight`:

```
member = property(member_getter, fdel=member_deleter)
```

Se você não usar uma propriedade, a remoção de um atributo também poderá ser tratada implementando o método especial de baixo nível `__delattr__` apresentado na seção “Métodos especiais para tratamento de atributos” na página 679. Implementar uma classe tola com `__delattr__` fica como exercício para o leitor com bastante tempo livre.

Propriedades são um recurso poderoso, mas, às vezes, alternativas mais simples e de mais baixo nível são preferíveis. Na última seção deste capítulo, veremos algumas APIs essenciais que Python oferece para programação de atributos dinâmicos.

Atributos e funções essenciais para tratamento de atributos

Neste capítulo, e mesmo antes no livro, usamos algumas funções embutidas e métodos especiais oferecidos por Python para lidar com atributos dinâmicos. Esta seção oferece uma visão geral deles em um só lugar, pois sua documentação está espalhada nos documentos oficiais.

Atributos especiais que afetam o tratamento de atributos

O comportamento de muitas funções e alguns métodos especiais listados nas seções a seguir dependem de três atributos especiais:

`__class__`

Uma referência à classe do objeto (por exemplo, `obj.__class__` é o mesmo que `type(obj)`). Python procura métodos especiais como `__getattr__` somente na classe de um objeto, e não nas instâncias.

`__dict__`

Um mapeamento que armazena os atributos de escrita de um objeto ou uma classe. Um objeto que tem um `__dict__` pode ter novos atributos definidos a qualquer momento. Se uma classe tiver um atributo `__slots__`, suas instâncias talvez não tenham um `__dict__`. Veja `__slots__` (a seguir).

`__slots__`

Um atributo que pode ser definido em uma classe para limitar os atributos que suas instâncias podem ter. `__slots__` é uma tuple de strings que nomeia os atributos permitidos.¹⁴ Se o nome `'__dict__'` não estiver em `__slots__`, as instâncias dessa classe não terão um `__dict__` próprio, e somente os atributos nomeados serão permitidos nessas instâncias.

Funções embutidas para tratamento de atributos

Essas cinco funções embutidas realizam leitura, escrita e introspecção em atributos de um objeto:

`dir([object])`

Lista a maior parte dos atributos do objeto. A documentação oficial (<http://bit.ly/1HGvLDV>) afirma que `dir` é destinado a uso interativo, portanto não oferece uma lista completa dos atributos, mas mostra um conjunto “interessante” de nomes. `dir` pode inspecionar objetos implementados com ou sem `__dict__`. O próprio atributo `__dict__` não é listado por `dir`, mas as chaves de `__dict__` são exibidas. Vários atributos especiais das classes, como `__mro__`, `__bases__` e `__name__` também não são listados por `dir`. Se o argumento opcional `object` não é fornecido, `dir` lista os nomes no escopo atual.

`getattr(object, name[, default])`

Obtém o atributo identificado pela string `name` de `object`. Pode devolver um atributo da classe do objeto ou de uma superclasse. Se esse atributo não existir, `getattr` levanta `AttributeError` ou devolve o valor `default`, se for especificado.

¹⁴ Alex Martelli destaca que, embora `__slots__` possa ser implementado como uma `list`, é melhor ser explícito e sempre usar uma `tuple`, pois mudar a lista em `__slots__` depois que o corpo da classe foi processado não tem efeito, portanto usar uma sequência mutável nesse caso seria confuso.

`hasattr(object, name)`

Devolve `True` se o atributo especificado existir em `object` ou, de algum modo, puder ser acessado por meio dele (por herança, por exemplo). A documentação (<https://docs.python.org/3/library/functions.html#hasattr>) explica o seguinte: “Essa função é implementada chamando `getattr(object, name)` e verificando se um `AttributeError` é ou não levantado.”

`setattr(object, name, value)`

Atribui `value` ao atributo especificado de `object`, se `object` permitir essa operação. Isso pode criar um novo atributo ou sobrescrever um atributo existente.

`vars([object])`

Devolve o `__dict__` de `object`; `vars` não é capaz de lidar com instâncias de classes que definam `__slots__` e não tenham um `__dict__` (compare com `dir`, que trata esse tipo de instância). Sem argumentos, `vars()` faz o mesmo que `locals()`: devolve um `dict` que representa o escopo local.

Métodos especiais para tratamento de atributos

Quando implementados em uma classe definida pelo usuário, os métodos especiais listados aqui tratam da recuperação, escrita, remoção e listagem de atributos.

Acesso a atributos usando a notação de ponto ou as funções embutidas `getattr`, `hasattr` e `setattr` dispara os métodos especiais correspondentes listados aqui. Ler e escrever em atributos diretamente no `__dict__` da instância não dispara esses métodos especiais – e essa é a maneira usual de evitá-los, se for necessário.

A seção “3.3.9. Special method lookup” (Busca de métodos especiais, <http://bit.ly/1cPO3qP>) do capítulo “Data model” (Modelo de dados) dá o seguinte aviso:

Para classes customizadas, garanta-se que chamadas implícitas a métodos especiais funcionarão corretamente somente se eles estiverem definidos no tipo do objeto, e não no dicionário da instância do objeto.

Em outras palavras, supõe-se que os métodos especiais serão recuperados da própria classe, mesmo quando o alvo da ação for uma instância. Por esse motivo, os métodos especiais não são encobertos pelos atributos de instância de mesmo nome.

Nos exemplos a seguir, suponha que haja uma classe chamada `Class`, `obj` seja uma instância de `Class` e `attr` seja um atributo de `obj`.

Para cada um desses métodos especiais, não importa se o acesso aos atributos seja feito com notação de ponto ou com uma das funções embutidas listadas na seção

“Funções embutidas para tratamento de atributos” na página 678. Por exemplo, tanto `obj.attr` quanto `getattr(obj, 'attr', 42)` disparam `Class.__getattr__(obj, 'attr')`.

`__delattr__(self, name)`

Sempre chamado quando há uma tentativa de apagar um atributo usando o comando `del`; por exemplo, `del obj.attr` dispara `Class.__delattr__(obj, 'attr')`.

`__dir__(self)`

Chamado quando `dir` é invocado no objeto para fornecer uma listagem dos atributos; por exemplo, `dir(obj)` dispara `Class.__dir__(obj)`.

`__getattr__(self, name)`

Chamado somente quando uma tentativa de recuperar o atributo especificado falha, depois de buscar em `obj`, `Class` e em suas superclasses. As expressões `obj.atr_inexistente`, `getattr(obj, 'atr_inexistente')` e `hasattr(obj, 'atr_inexistente')` podem disparar `Class.__getattr__(obj, 'atr_inexistente')`, mas somente se um atributo com esse nome não puder ser encontrado em `obj` ou em `Class` e em suas superclasses.

`__getattribute__(self, name)`

Sempre chamado quando há uma tentativa de recuperar o atributo especificado, exceto quando o atributo buscado é um atributo especial ou método especial. A notação de ponto e as funções embutidas `getattr` e `hasattr` disparam esse método. `__getattr__` é chamado somente depois de `__getattribute__`, e apenas quando `__getattribute__` levanta `AttributeError`. Para recuperar atributos da instância `obj` sem disparar uma recursão infinita, implementações de `__getattribute__` devem usar `super().__getattribute__(obj, name)`.

`__setattr__(self, name, value)`

Sempre chamado quando há uma tentativa de escrever no atributo especificado. A notação de ponto e a função embutida `setattr` disparam esse método; por exemplo, tanto `obj.attr = 42` quanto `setattr(obj, 'attr', 42)` disparam `Class.__setattr__(obj, 'attr', 42)`.



Na prática, por serem incondicionalmente chamados e afetarem praticamente todo acesso a atributos, os métodos especiais `__getattribute__` e `__setattr__` são mais difíceis de usar corretamente que `__getattr__` – que só trata nomes inexistentes de atributos. Usar propriedades ou descritores é menos suscetível a erros que definir esses métodos especiais.

Com isso, concluímos nossa exploração de propriedades, métodos especiais e outras técnicas para implementar atributos dinâmicos.

Resumo do capítulo

Começamos nossa discussão sobre atributos dinâmicos mostrando exemplos práticos de classes simples para lidar facilmente com uma massa de dados JSON. O primeiro exemplo foi a classe `FrozenJSON`, que convertia dicionários e listas aninhados em instâncias aninhadas de `FrozenJSON` e em listas desse tipo de objeto. O código de `FrozenJSON` mostrou o uso do método especial `__getattr__` para converter estruturas de dados durante a execução sempre que seus atributos eram lidos. A última versão de `FrozenJSON` mostrou o uso do método construtor `__new__` para transformar uma classe em uma fábrica flexível de objetos, não limitada às instâncias da própria classe.

Em seguida, convertemos o feed JSON em um banco de dados `shelve.Shelf` armazenando instâncias serializadas de uma classe `Record`. A primeira versão de `Record` tinha algumas linhas e apresentou a técnica do “punhado” (bunch): usar `self.__dict__.update(**kwargs)` para criar atributos arbitrários a partir de argumentos nomeados passados para `__init__`. A segunda iteração desse exemplo mostrou a extensão de `Record` com uma classe `DbRecord` para integração do banco de dados e uma classe `Event` que implementa a recuperação automática de registros associados usando propriedades.

A discussão sobre propriedades continuou com a classe `LineItem`, em que uma propriedade foi implementada para evitar que o atributo `weight` recebesse valores negativos ou zero, por não fazerem sentido no contexto da aplicação. Após um olhar detalhado sobre a sintaxe e a semântica das propriedades, criamos uma fábrica de propriedades para garantir a mesma validação em `weight` e em `price` sem implementar vários getters e setters. A fábrica de propriedades aproveitou conceitos sutis – como closures e o encobrimento de atributos de instância pelas propriedades – para oferecer uma solução genérica elegante usando o mesmo número de linhas que uma única definição de propriedade feita manualmente.

Por fim, vimos rapidamente o tratamento da remoção de atributos com propriedades, seguido de uma visão geral dos principais atributos especiais, funções embutidas e métodos especiais que dão suporte à metaprogramação de atributos no núcleo da linguagem Python.

Leituras complementares

A documentação oficial das funções embutidas para tratamento de atributos e introspecção está no capítulo 2, “Built-in Functions” (Funções embutidas, <http://bit.ly/lcPOrpc>) de *The Python Standard Library* (Biblioteca-padrão de Python). Os métodos especiais relacionados e o atributo especial `__slots__` estão documentados em *The Python Language*

Reference (Guia de referência à linguagem Python) na seção “3.3.2. Customizing attribute access” (Customizando o acesso a atributos, <http://bit.ly/1cPOlxV>). A semântica de como os métodos especiais são chamados ignorando as instâncias é explicada na seção “3.3.9. Special method lookup” (Busca de métodos especiais, <http://bit.ly/1cPO3qP>). No capítulo 4, “Built-in Types” (Tipos embutidos) de *The Python Standard Library* (Biblioteca-Padrão de Python), a seção “4.13. Special Attributes” (Atributos especiais, <http://bit.ly/1cPOodb>) inclui os atributos `__class__` e `__dict__`.

O livro *Python Cookbook*, 3E de David Beazley e Brian K. Jones (O’Reilly, <http://shop.oreilly.com/product/0636920027072.do>)¹⁵ tem várias receitas que incluem os assuntos deste capítulo, mas citarei três que se destacam: “Recipe 8.8. Extending a Property in a Subclass” (Estender uma propriedade em uma subclasse) aborda a complicada questão de sobrescrever os métodos em uma propriedade herdada de uma superclasse, “Recipe 8.15. Delegating Attribute Access” (Delegar acesso a atributos) implementa uma classe proxy que exhibe a maioria dos métodos especiais da seção “Métodos especiais para tratamento de atributos” na página 679 deste livro e “Recipe 9.21. Avoiding Repetitive Property Methods” (Evitar métodos repetitivos de propriedades), que serviu de base para a fábrica de propriedades apresentada no exemplo 19.24.

O livro *Python in a Nutshell*, 2E (O’Reilly, <http://shop.oreilly.com/product/9780596100469.do>) de Alex Martelli cobre somente Python 2.5, mas os fundamentos continuam aplicáveis em Python 3, e sua abordagem é rigorosa e objetiva. Martelli dedica apenas três páginas às propriedades, mas isso é porque o livro segue um estilo de apresentação axiomático: as quinze páginas anteriores oferecem uma descrição completa da semântica das classes Python desde o princípio, incluindo descritores – o modo como as propriedades são realmente implementadas internamente. Assim, quando chega nas propriedades, Martelli consegue reunir muitos insights nessas três páginas – incluindo aquele que escolhi para iniciar este capítulo.

Bertrand Meyer, autor da citação na definição do *Princípio de Acesso Uniforme* na abertura deste capítulo, escreveu o excelente livro *Object-Oriented Software Construction*, 2E (Prentice-Hall). O livro tem mais de 1.250 páginas, e confesso que não o li por inteiro, mas os seis primeiros capítulos oferecem uma das melhores introduções conceituais à análise e ao design orientados a objetos que já vi; o capítulo 11 apresenta *Design by Contract* (Design por contrato – Meyer criou o método e cunhou o termo) e o capítulo 35 apresenta suas avaliações de algumas linguagens orientadas a objetos fundamentais: Simula, Smalltalk, CLOS (a extensão orientada a objetos de Lisp), Objective-C, C++ e Java, além de comentários rápidos sobre outras linguagens. Meyer também é o inventor do pseudo-pseudocódigo: somente na última página do livro ele revela que a “notação” usada em todo o livro como pseudocódigo é, na verdade, a linguagem Eiffel.

15 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

Ponto de vista

O *Princípio do Acesso Uniforme* (Uniform Access Principle, às vezes chamado de UAP pelos amantes de siglas) de Meyer é esteticamente atraente. Como programador que usa uma API, eu não deveria me importar se `coconut.price` simplesmente busca um atributo de dados ou faz algum cálculo. Como consumidor e cidadão, eu me importo: em e-commerce atualmente, o valor de `coconut.price` muitas vezes depende de quem está perguntando; portanto, certamente não é apenas um atributo de dados. De fato, é uma prática comum fornecer um preço menor se a consulta vier de fora da loja – por exemplo, de uma ferramenta de comparação de preços. Essa prática pune consumidores leais que gostam de navegar em uma loja em particular. Mas estou apenas divagando.

Entretanto a divagação anterior levanta um ponto relevante em programação: embora o Princípio do Acesso Uniforme faça todo o sentido em um mundo ideal, na realidade, os usuários de uma API talvez precisem saber se a leitura de `coconut.price` pode ser possivelmente custosa ou consumir muito tempo. Como é de esperar em se tratando de engenharia de software, a Wiki original (<http://bit.ly/1HGvZuA>) de Ward Cunningham contém argumentos perspicazes sobre os méritos do Princípio do Acesso Uniforme (<http://bit.ly/1HGvNvk>).

Em linguagens de programação orientadas a objetos, a aplicação ou as violações ao Princípio do Acesso Uniforme normalmente estão em torno da sintaxe da leitura de atributos de dados públicos em comparação com chamadas a métodos `getter/setter`.

Smalltalk e Ruby abordam essa questão de maneira simples e elegante: elas simplesmente não têm suporte para atributos de dados públicos. Todo atributo de instância nessas linguagens é privado, portanto todo acesso a eles precisa ocorrer por meio de métodos. Porém sua sintaxe deixa essa operação simples: em Ruby, `coconut.price` chama o método `price`; em Smalltalk, basta escrever `coconut price`.

Na outra extremidade do espectro, a linguagem Java permite ao programador escolher entre quatro modificadores de níveis de acesso.¹⁶ Contudo a prática generalizada está em desacordo com a sintaxe definida pelos projetistas de Java. No reino de Java, todos concordam que os atributos deveriam ser `private`, e é obrigado a escrever essa palavra sempre, pois não é o default. Quando todos os atributos são privados, todo acesso a eles feito de fora da classe tem de ser por meio de métodos de acesso. As IDEs de Java incluem atalhos para gerar métodos de acesso de modo automático. Infelizmente, a IDE não é tão útil assim quando

¹⁶ Incluindo o default sem nome que o Tutorial de Java (<http://bit.ly/1cPOMIE>) chama de “package-private” (privado ao pacote).

precisamos ler o código seis meses depois. Cabe a você vasculhar uma infinidade de métodos de acesso que não fazem nada a fim de encontrar aqueles que são importantes por implementarem alguma lógica de negócios.

Alex Martelli fala pela maioria da comunidade Python quando chama os métodos de acesso de “goofy idioms” (idioms tolos) e então apresenta os exemplos a seguir, que parecem bem diferentes, mas fazem a mesma coisa: ¹⁷

```
someInstance.widgetCounter += 1
# em vez de...
someInstance.setWidgetCounter(someInstance.getWidgetCounter() + 1)
```

Às vezes, ao projetar APIs, fico me perguntando se todo método que não recebe um argumento (além de `self`), devolve um valor (diferente de `None`) e é uma função pura (isto é, não tem efeitos colaterais) não deveria ser substituído por uma propriedade somente de leitura. Neste capítulo, o método `LineItem.subtotal` (como no exemplo 19.23) seria um bom candidato a se tornar uma propriedade somente de leitura. É claro que isso exclui métodos projetados para alterar o objeto, como `my_list.clear()`. Seria uma péssima ideia transformá-lo em uma propriedade, pois um simples acesso a `my_list.clear` poderia apagar o conteúdo da lista!

Na biblioteca Pingo.io para programação GPIO (<http://www.pingo.io/docs/> – o projeto foi mencionado na seção “Método `__missing__`” na página 102), boa parte da API no nível de usuário é baseada em propriedades. Por exemplo, para ler o valor atual de um pino analógico, o usuário escreve `pin.value`, e configurar o modo de um pino digital é escrito como `pin.mode = OUT`. Internamente, ler o valor de um pino analógico ou configurar o modo de um pino digital pode envolver bastante código, dependendo do driver específico da placa. Decidimos usar propriedades em Pingo porque queríamos que a API fosse fácil de usar mesmo em ambientes interativos como em iPython Notebook (<http://ipython.org/notebook.html>), e achamos que `pin.mode = OUT` é mais legível e mais fácil de digitar que `pin.set_mode(OUT)`.

Embora ache a solução de Smalltalk e de Ruby mais limpa, penso que a abordagem de Python faz mais sentido que a de Java. Podemos começar de forma simples, implementando membros de dados como atributos públicos, pois sabemos que eles sempre poderão ser encapsulados por propriedades (ou por descritores, sobre os quais falaremos no próximo capítulo).

`__new__` é melhor que `new`

Outro exemplo do Princípio do Acesso Uniforme (ou de uma variação dele) é o fato de chamadas a função e instanciação de objetos usarem a mesma sintaxe em Python: `my_obj = foo()`, em que `foo` pode ser uma classe ou qualquer outro invocável.

¹⁷ Alex Martelli, *Python in a Nutshell*, 2E (O'Reilly), p. 101.

Outras linguagens influenciadas pela sintaxe de C++ têm um operador `new` que faz a instanciação parecer diferente de uma chamada de função. Na maioria das vezes, para o usuário de uma API, não importa se `foo` é uma função ou uma classe. Até recentemente, eu achava que `property` era uma função. Em uso normal, não faz diferença.

Há muitos bons motivos para substituir construtores por fábricas.¹⁸ Um motivo popular é limitar o número de instâncias, devolvendo instâncias criadas anteriormente (como no padrão Singleton). Um uso relacionado a esse é usar um cache para reduzir processos custosos de construção de objetos. Além disso, às vezes é conveniente devolver objetos de tipos diferentes, de acordo com os argumentos especificados.

Implementar um construtor é mais simples; fornecer uma fábrica acrescenta flexibilidade às custas de mais código. Em linguagens que têm um operador `new`, o projetista de uma API precisa decidir previamente se vai ficar com um construtor simples ou investir em uma fábrica. Se a escolha inicial for errada, a correção pode sair caro – tudo porque `new` é um operador.

Às vezes, também pode ser conveniente ir na outra direção e substituir uma função simples por uma classe.

Em Python, classes e funções são intercambiáveis em muitas situações. Não só porque não há um operador `new`, mas também porque existe um método especial `__new__`, que pode transformar uma classe em uma fábrica produzindo objetos de tipos diferentes (como vimos na seção “Criação flexível de objetos com `__new__`” na página 652) ou devolvendo instâncias previamente criadas em vez de criar uma nova instância toda vez.

Seria mais fácil aproveitar essa dualidade função-classe se a *PEP 8 – Style Guide for Python Code* (Guia de estilo para código Python, <http://bit.ly/1HGvYH7>) não recomendasse `CamelCase` para nomes de classe. Por outro lado, dezenas de classes da biblioteca-padrão usam nomes com letras minúsculas (por exemplo, `property`, `str`, `defaultdict` etc.). Talvez o uso de nomes de classe com letras minúsculas seja uma virtude, e não um vício. Contudo, independentemente de como encaramos isso, o uso inconsistente de maiúsculas em classes na biblioteca-padrão de Python representa um problema de usabilidade.

¹⁸ Os motivos que estou prestes a mencionar estão no artigo de Dr. Dobbs Journal cujo título é “Java’s `new` Considered Harmful” (`new` de Java é considerado prejudicial, <http://ubm.io/1cPP4PN>) de Jonathan Amsterdam e em “Consider static factory methods instead of constructors” (Considere métodos estáticos de fábrica em vez de construtores), que é o item 1 do livro premiado *Effective Java* (Addison-Wesley) de Joshua Bloch.

Embora chamar uma função não seja diferente de chamar uma classe, é bom saber qual é qual, por causa de outra operação que podemos fazer com uma classe: criar subclasses. Sendo assim, pessoalmente, uso `CamelCase` em todas as classes que escrevo, e gostaria que todas as classes da biblioteca-padrão de Python usassem a mesma convenção. Estou olhando para vocês, `collections.OrderedDict` e `collections.defaultdict`.

CAPÍTULO 20

Descritores de atributos

Conhecer descritores não só possibilita o acesso a um conjunto maior de ferramentas como também proporciona uma compreensão mais profunda de como Python funciona e uma apreciação da elegância de seu design.¹

— Raymond Hettinger
Core developer e guru de Python

Descritores são uma maneira de reutilizar a mesma lógica de acesso em vários atributos. Por exemplo, tipos de campos em ORMs, como o ORM de Django ou o SQLAlchemy, são descritores que gerenciam o fluxo de dados dos campos de um registro do banco de dados para atributos de um objeto Python, e vice-versa.

Um descritor é uma classe que implementa um protocolo constituído pelos métodos `__get__`, `__set__` e `__delete__`. A classe `property` implementa o protocolo completo de descritores. Como normalmente ocorre com protocolos, implementações parciais não são um problema. De fato, a maioria dos descritores que vemos em códigos de verdade implementa apenas `__get__` e `__set__`, e muitos implementam apenas um desses métodos.

Descritores são um recurso característico de Python e são usados não apenas em aplicações, mas também na infraestrutura da linguagem. Além de propriedades, outros recursos de Python que aproveitam os descritores são os métodos de uma forma geral, bem como os decoradores `classmethod` e `staticmethod`. Entender os descritores é fundamental para dominar a linguagem Python. É esse o assunto deste capítulo.

Exemplo de descritor: validação de atributos

Como vimos na seção “Implementando uma fábrica de propriedades” na página 673, uma fábrica de propriedades é uma maneira de evitar códigos repetitivos para getters

¹ Raymond Hettinger, *Descriptor How To Guide* (HowTo para descritores, <https://docs.python.org/3/howto/descriptor.html>).

e setters aplicando padrões de programação funcional. Uma fábrica de propriedades é uma função de ordem superior que cria um conjunto parametrizado de funções de acesso e constrói uma instância customizada de property usando estas funções, com closures para armazenar configurações como `storage_name`. O modo orientado a objetos de resolver o mesmo problema é usar uma classe descritora (descriptor class). Continuaremos com a série de exemplos de `LineItem` no ponto em que paramos na seção “Implementando uma fábrica de propriedades” na página 673, refatorando a fábrica de propriedades `quantity` em uma classe descritora `Quantity`.

LineItem tomada #3: um descritor simples

Uma classe que implementa um método `__get__`, `__set__` ou `__delete__` é um descritor. Usa-se um descritor declarando instâncias dele como atributos de classe em uma outra classe. Criaremos um descritor `Quantity`, e a classe `LineItem` usará duas instâncias de `Quantity`: uma para gerenciar o atributo `weight` e outra para `price`. Um diagrama ajuda, portanto dê uma olhada na figura 20.1.

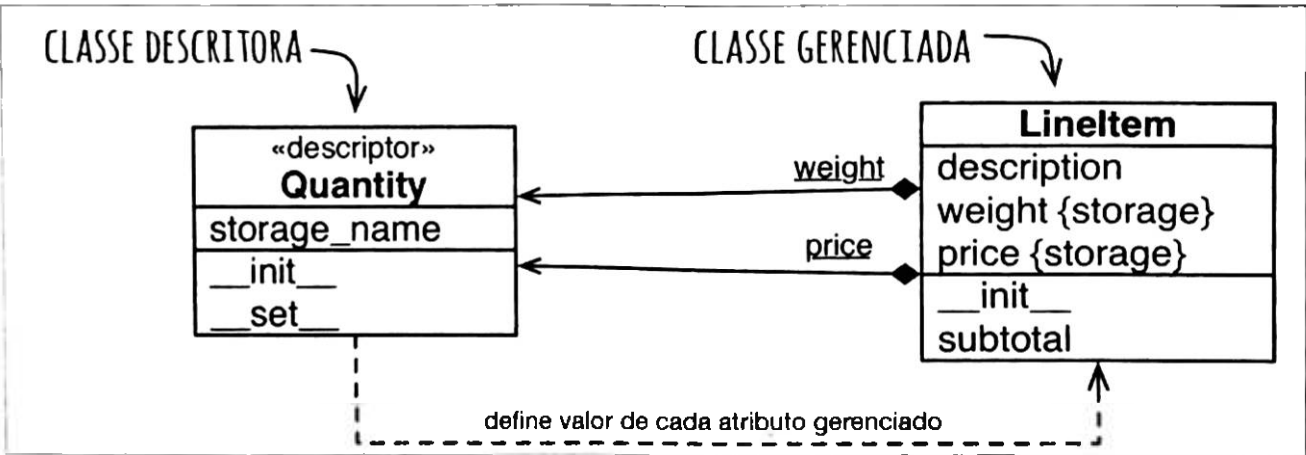


Figura 20.1 – Diagrama de classes UML para `LineItem` usando uma classe descritora chamada `Quantity`. Atributos sublinhados em UML são atributos de classe. Observe que `weight` e `price` são instâncias de `Quantity` associadas à classe `LineItem`, mas instâncias de `LineItem` também têm seus próprios atributos `weight` e `price`, onde esses valores são armazenados.

Observe que a palavra `weight` aparece duas vezes na figura 20.1 porque, na verdade, há dois atributos distintos de nome `weight`: um é um atributo de classe de `LineItem`, o outro é um atributo de instância presente em cada objeto `LineItem`. O mesmo se aplica a `price`.

A partir de agora, usarei as seguintes definições:

Classe descritora

Uma classe que implementa o protocolo de descritor. É a classe `Quantity` na figura 20.1.

Classe gerenciada

A classe em que as instâncias do descritor são declaradas como atributos de classe – `LineItem` na figura 20.1.

Instância do descritor

Cada instância de uma classe descritora, declarada como um atributo de classe na classe gerenciada. Na figura 20.1, cada instância do descritor é representada por uma seta de composição com um nome sublinhado (sublinhado quer dizer um atributo de classe em UML). Os losangos pretos tocam a classe `LineItem`, que contém as instâncias do descritor.

Instância gerenciada

Uma instância da classe gerenciada. Nesse exemplo, instâncias de `LineItem` serão as instâncias gerenciadas (não aparecem no diagrama de classes).

Atributo de armazenagem

Um atributo da instância gerenciada que armazena o valor de um atributo gerenciado para essa instância em particular. Na figura 20.1, os atributos de instância `weight` e `price` de `LineItem` são atributos de armazenagem. São diferentes das instâncias do descritor, que são sempre atributos da classe.

Atributo gerenciado

Um atributo público na classe gerenciada que será tratado por uma instância do descritor, com valores mantidos em atributos de armazenagem. Em outras palavras, uma instância de descritor e um atributo de armazenagem oferecem a infraestrutura para um atributo gerenciado.

É importante perceber que instâncias de `Quantity` são atributos de classe de `LineItem`. Esse ponto crucial é enfatizado pelos “mills and gizmos” (engenhocas e bugigangas) na figura 20.2.

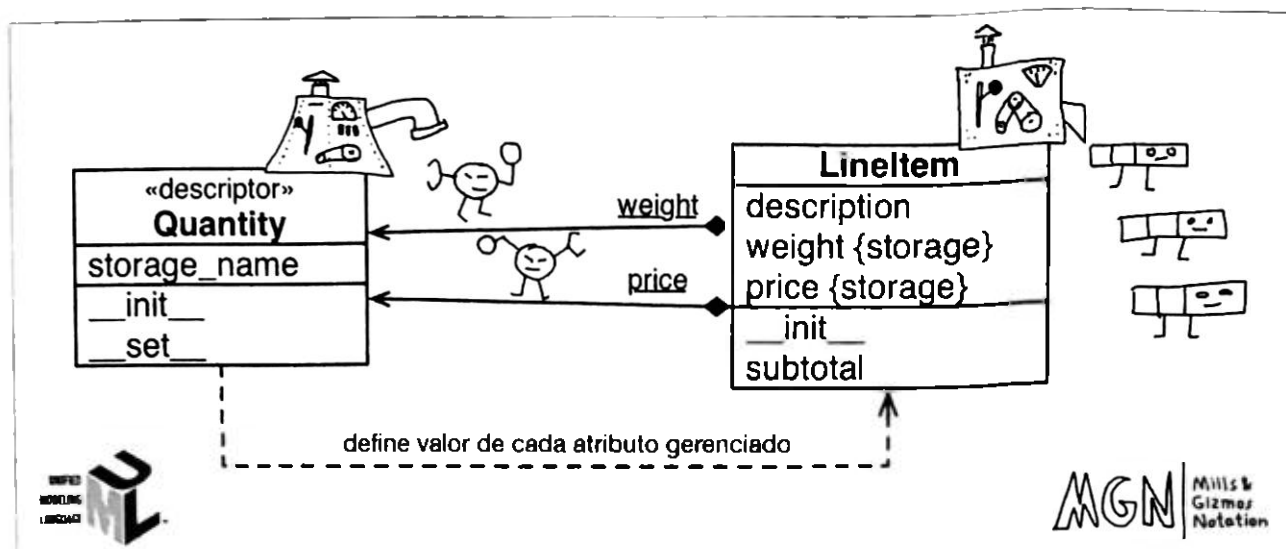


Figura 20.2 – Diagrama de classes UML com anotações usando MGN (Mills & Gizmos Notation): classes são engenhocas (mills) que produzem bugigangas (gizmos) – as instâncias. A engenhoca *Quantity* produz duas bugigangas vermelhas, que são associados à engenhoca *LineItem*: *weight* e *price*. A engenhoca *LineItem* produz bugigangas azuis que têm seus próprios atributos *weight* e *price*, em que esses valores serão armazenados.

Introdução à Mills & Gizmos Notation

Depois de explicar descritores muitas vezes, percebi que UML não é muito bom para mostrar relacionamentos que envolvam classes e instâncias, como o relacionamento entre uma classe gerenciada e as instâncias de descritor.² Então inventei minha própria “linguagem” – MGN (Mills & Gizmos Notation) –, que uso para anotações em diagramas UML.

MGN foi projetada para deixar bem clara a distinção entre classes e instâncias. Veja a figura 20.3. Em MGN, uma classe é desenhada como uma “engenhoca” (mill), uma máquina complicada que produz bugigangas (gizmos). Classes/engenhocas são sempre máquinas com alavancas e mostradores. As bugigangas são as instâncias, e são bem mais simples. Uma bugiganga tem a mesma cor da engenhoca que a produziu.

² Classes e instâncias são desenhadas como retângulos em diagramas de classe UML. Há diferenças no visual, mas instâncias raramente são mostradas em diagramas de classes, portanto talvez os desenvolvedores não as reconheçam.



Figura 20.3 – Esboço da MGN mostrando a classe `LineItem` criando três instâncias e `Quantity` criando duas. Uma instância de `Quantity` está recuperando um valor armazenado em uma instância de `LineItem`.

Nesse exemplo, desenhei instâncias de `LineItem` como linhas de um pedido de compra, com três células representando os três atributos (`description`, `weight` e `price`). Como instâncias de `Quantity` são descritores, eles têm uma lente de aumento para obter valores (com `__get__`) e uma garra para alterar valores (com `__set__`). Quando chegarmos às metaclasses, você me agradecerá por esses rabiscos.

Chega de rabiscos por enquanto. Vamos ao código: o exemplo 20.1 mostra a classe descritora `Quantity` e uma nova classe `LineItem` que usa duas instâncias de `Quantity`.

Exemplo 20.1 – `bulkfood_v3.py`: descritores `Quantity` administram atributos em `LineItem`

```
class Quantity: ❶
    def __init__(self, storage_name):
        self.storage_name = storage_name ❷
    def __set__(self, instance, value): ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value ❹
        else:
            raise ValueError('value must be > 0')

class LineItem:
    weight = Quantity('weight') ❺
    price = Quantity('price') ❻
    def __init__(self, description, weight, price): ❼
        self.description = description
        self.weight = weight
        self.price = price
```

```
def subtotal(self):
    return self.weight * self.price
```

- ❶ O mecanismo de descritores é baseado em protocolo; não é preciso herdar de alguma classe especial para criar um descritor.
- ❷ Cada instância de `Quantity` terá um atributo `storage_name`: é o nome do atributo que armazenará o valor nas instâncias gerenciadas.
- ❸ `__set__` é chamado quando há uma tentativa de escrever um valor em um atributo gerenciado. Nesse caso, `self` é a instância do descritor (ou seja, `LineItem.weight` ou `LineItem.price`), `instance` é a instância gerenciada (uma instância de `LineItem`) e `value` é o valor atribuído.
- ❹ Aqui precisamos usar diretamente o `__dict__` da instância gerenciada; tentar usar a função embutida `setattr` dispararia o método `__set__` novamente, resultando em uma recursão infinita.
- ❺ A primeira instância do descritor é vinculada ao atributo `weight`.
- ❻ A segunda instância do descritor é vinculada ao atributo `price`.
- ❼ O restante do corpo da classe é tão simples e limpo quanto o código original em *bulkfood_v1.py* (Exemplo 19.15).

No exemplo 20.1, cada atributo gerenciado tem o mesmo nome de seu atributo de armazenagem, e não há uma lógica especial para `getter`, portanto `Quantity` não precisa de um método `__get__`.

O código do exemplo 20.1 funciona conforme esperado, evitando a venda de trufas a um preço igual a zero:³

```
>>> truffle = LineItem('White truffle', 100, 0)
```

```
Traceback (most recent call last):
```

```
...
```

```
ValueError: value must be > 0
```



Ao implementar um método `__set__`, tenha em mente o que os argumentos `self` e `instance` representam: `self` é a instância do descritor e `instance` é a instância gerenciada. Descritores que gerenciam atributos de instância devem armazenar valores nas instâncias gerenciadas. É por isso que Python oferece o argumento `instance` aos métodos de descritor.

3 Trufas brancas custam milhares de dólares o quilo. Impedir a venda de trufas a 1 centavo fica como exercício para o leitor empreendedor. Conheço uma pessoa que comprou uma enciclopédia de estatística de 1.800 dólares por 18 dólares por causa de um erro em uma loja online (não é a Amazon.com).

Pode ser tentador armazenar o valor de cada atributo gerenciado na própria instância do descritor, mas está errado. Em outras palavras, no método `__set__`, em vez de escrever:

```
instance.__dict__[self.storage_name] = value
```

a alternativa tentadora, porém ruim, seria:

```
self.__dict__[self.storage_name] = value
```

Para entender por que isso seria errado, pense no que os dois primeiros argumentos de `__set__` querem dizer: `self` e `instance`. Nesse caso, `self` é a instância do descritor, que, na verdade, é um atributo de classe da classe gerenciada. Você pode ter milhares de instâncias de `LineItem` na memória ao mesmo tempo, mas terá apenas duas instâncias de descritores: `LineItem.weight` e `LineItem.price`. Portanto tudo que você armazenar nas próprias instâncias do descritor, na realidade, fará parte de um atributo de classe de `LineItem` e, sendo assim, será compartilhado entre todas as instâncias de `LineItem`.

Uma desvantagem do exemplo 20.1 é a necessidade de repetir os nomes dos atributos quando os descritores são instanciados no corpo da classe gerenciada. Seria bom se a classe `LineItem` pudesse ser declarada assim:

```
class LineItem:
    weight = Quantity()
    price = Quantity()

    # métodos restantes sem alteração
```

O problema é que – como vimos no capítulo 8 – o lado direito de uma atribuição é executado antes de a variável existir. A expressão `Quantity()` é avaliada para criar uma instância do descritor e, nesse momento, o código da classe `Quantity` não tem como adivinhar o nome da variável ao qual o descritor será vinculado (por exemplo, `weight` ou `price`).

Como está, o exemplo 20.1 exige que cada `Quantity` seja nomeado explicitamente, o que não só é inconveniente como também perigoso: se um programador copiar e colar o código, esquecer-se de editar os nomes e escrever algo como `price = Quantity('weight')`, o programa se comportará muito mal, sobrescrevendo o valor de `weight` sempre que escrever em `price`.

Uma solução não tão elegante, porém funcional, para o problema do nome repetido será apresentada a seguir. Soluções melhores exigem um decorador de classe ou uma metaclasses, portanto serão deixadas para o capítulo 21.

LineItem tomada #4: nomes automáticos para atributos de armazenagem

Para evitar digitar novamente o nome do atributo nas declarações do descritor, vamos gerar uma string única para o `storage_name` de cada instância de `Quantity`. A figura 20.4 mostra o diagrama UML atualizado para as classes `Quantity` e `LineItem`.

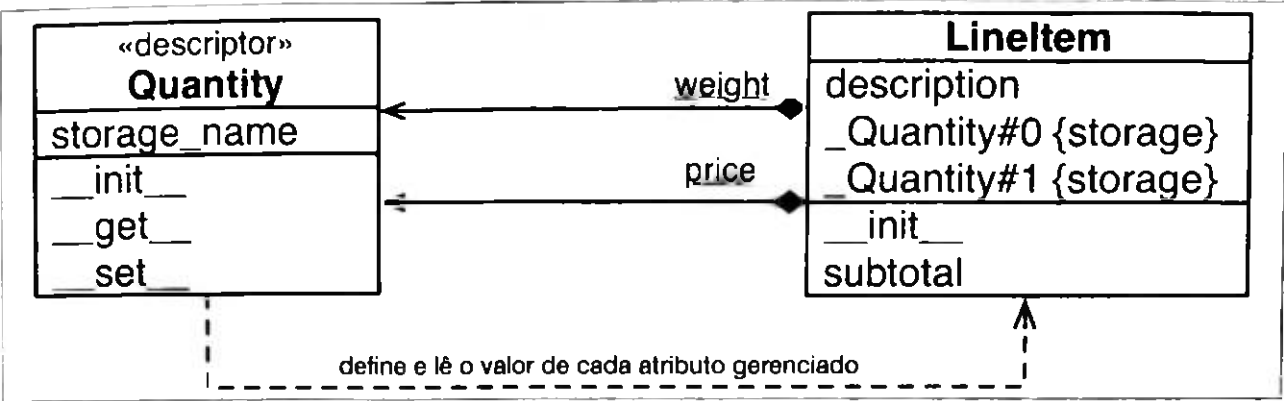


Figura 204 – Diagrama UML para o exemplo 20.2. Agora Quantity tem métodos get e set e instâncias de LineItem têm atributos de armazenagem com nomes gerados: _Quantity#0 e _Quantity#1.

Para gerar storage_name, começamos com um prefixo '_Quantity#' e concatenamos um inteiro: o valor atual de um atributo de classe Quantity.__counter que incrementaremos sempre que uma nova instância do descritor Quantity for associada a uma classe. Usar o caractere de sustenido no prefixo garante que storage_name não entrará em conflito com atributos criados pelo usuário utilizando a notação de ponto, pois nutmeg._Quantity#0 não é uma sintaxe válida em Python. Mas sempre podemos ler e escrever em atributos com identificadores “inválidos” como esses usando as funções embutidas getattr e setattr, ou acessando o __dict__ da instância. O exemplo 20.2 mostra a nova implementação.

Exemplo 20.2 – bulkfood_v4.py: cada descritor Quantity obtém um storage_name único

```
class Quantity:
    __counter = 0 ❶
    def __init__(self):
        cls = self.__class__ ❷
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '{}#{}'.format(prefix, index) ❸
        cls.__counter += 1 ❹
    def __get__(self, instance, owner): ❺
        return getattr(instance, self.storage_name) ❻
    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value) ❼
        else:
            raise ValueError('value must be > 0')
```

```
class LineItem:
    weight = Quantity() ❹
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

- ❶ `__counter` é um atributo de classe de `Quantity` que conta o número de instâncias de `Quantity`.
- ❷ `cls` é uma referência à classe `Quantity`.
- ❸ O `storage_name` de cada instância do descritor é único porque é criado a partir do nome da classe descritora e do valor atual de `__counter` (por exemplo, `_Quantity#0`).
- ❹ Incrementa `__counter`.
- ❺ Precisamos implementar `__get__` porque o nome do atributo gerenciado não é o mesmo de `storage_name`. O argumento `owner` será explicado em breve.
- ❻ Usa a função embutida `getattr` para recuperar o valor de `instance`.
- ❼ Usa a função embutida `setattr` para armazenar o valor em `instance`.
- ❽ Agora não precisamos passar o nome do atributo gerenciado ao construtor `Quantity`. Esse era o objetivo dessa versão.

Nesse caso, podemos usar as funções embutidas `getattr` e `setattr` de nível mais alto para armazenar o valor – em vez de recorrer a `instance.__dict__` – porque o atributo gerenciado e o atributo de armazenagem têm nomes diferentes, portanto chamar `getattr` no atributo de armazenagem não disparará o descritor, evitando a recursão infinita discutida no exemplo 20.1.

Se você testar `bulkfood_v4.py`, verá que os descritores `weight` e `price` funcionam como esperado, e os atributos de armazenagem também podem ser lidos diretamente, o que é útil para depuração:

```
>>> from bulkfood_v4 import LineItem
>>> coconuts = LineItem('Brazilian coconut', 20, 17.95)
>>> coconuts.weight, coconuts.price
(20, 17.95)
>>> getattr(raisins, '_Quantity#0'), getattr(raisins, '_Quantity#1')
(20, 17.95)
```



Se quiséssemos seguir a convenção de Python para desfiguração de nomes (name mangling) – por exemplo, `_LineItem__quantity0` –, teríamos de saber o nome da classe gerenciada (ou seja, `LineItem`), mas o corpo de uma definição de classe executa antes de a própria classe ser construída pelo interpretador, portanto não temos essa informação quando a instância de cada descritor é criada. No entanto, nesse caso, não é preciso incluir o nome da classe gerenciada para evitar a sobrescrita acidental desses atributos em subclasses: o `__counter` da classe descritora será incrementado sempre que um novo descritor for instanciado, garantindo que o nome de cada atributo de armazenagem seja único entre todas as classes gerenciadas.

Observe que `__get__` recebe três argumentos: `self`, `instance` e `owner`. O argumento `owner` é uma referência à classe gerenciada (por exemplo, `LineItem`), e é conveniente quando o descritor é usado para ler atributos da classe. Se um atributo gerenciado, como `weight`, é recuperado por meio da classe, por exemplo, `LineItem.weight`, o método `__get__` do descritor recebe `None` como valor do argumento `instance`. Isso explica o `AttributeError` na sessão de console a seguir:

```
>>> from bulkfood_v4 import LineItem
>>> LineItem.weight
Traceback (most recent call last):
...
File ".../descriptors/bulkfood_v4.py", line 54, in __get__
    return getattr(instance, self.storage_name)
AttributeError: 'NoneType' object has no attribute '_Quantity#0'
```

Levantar `AttributeError` é uma opção quando implementamos `__get__`, mas, se você optar por isso, a mensagem deverá ser corrigida para eliminar a menção confusa a `NoneType` e `_Quantity#0`, que são detalhes de implementação. Uma mensagem melhor seria "'LineItem' class has no such attribute" (Classe 'LineItem' não tem esse atributo). O ideal seria que o nome do atributo ausente fosse exibido, mas o descritor não sabe o nome do atributo gerenciado nesse exemplo, portanto não podemos fazer nada melhor por enquanto.

Por outro lado, para dar suporte à introspecção e a outros truques de metaprogramação dos usuários de sua classe, é uma boa prática fazer `__get__` devolver o próprio objeto descritor quando o atributo gerenciado é acessado por meio da classe. O exemplo 20.3 é uma pequena variação do exemplo 20.2, adicionando um pouco de lógica a `Quantity.__get__`.

Exemplo 20.3 – `bulkfood_v4b.py` (listagem parcial): quando chamado por meio da classe gerenciada, `get` devolve uma referência ao próprio descritor

```
class Quantity:
    __counter = 0

    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        if instance is None:
            return self ❶
        else:
            return getattr(instance, self.storage_name) ❷

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)
        else:
            raise ValueError('value must be > 0')
```

❶ Se a chamada não foi feita por meio de uma instância, devolve o próprio descritor.

❷ Caso contrário, devolve o valor do atributo gerenciado, como é feito normalmente.

Ao testar o exemplo 20.3, você verá o seguinte:

```
>>> from bulkfood_v4b import LineItem
>>> LineItem.price
<bulkfood_v4b.Quantity object at 0x100721be0>
>>> br_nuts = LineItem('Brazil nuts', 10, 34.95)
>>> br_nuts.price
34.95
```

Observando o exemplo 20.2, você pode achar que é muito código apenas para gerenciar dois atributos, mas é importante perceber que a lógica do descritor agora foi abstraída em uma unidade de código separada: a classe `Quantity`. Normalmente, não definimos um descritor no mesmo módulo em que ele é usado, mas em um módulo utilitário separado, projetado para ser usado em toda a aplicação – até mesmo em várias aplicações, se você estiver desenvolvendo um framework.

Com isso em mente, o exemplo 20.4 representa melhor o uso típico de um descritor.

Exemplo 20.4 – `bulkfood_v4c.py`: definição mais leve de `LineItem`; a classe descritora `Quantity` agora está no módulo importado `model_v4c`

```
import model_v4c as model ❶

class LineItem:
    weight = model.Quantity() ❷
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Importa o módulo `model_v4c` dando-lhe um nome mais amigável.

❷ Coloca `model.Quantity` em uso.

Usuários de Django perceberão que o exemplo 20.4 se parece muito com uma definição de modelo. Não é coincidência: campos de modelos em Django são descritores.



Conforme implementado até agora, o descritor `Quantity` funciona muito bem. O único inconveniente é usarmos identificadores de armazenagem gerados, por exemplo, `_Quantity#0`, dificultando a depuração para os usuários. Porém atribuir automaticamente nomes de armazenagem que lembrem os nomes dos atributos gerenciados exige um decorador de classe ou uma metaclasses, assuntos que adiaremos até o capítulo 21.

Como descritores são definidos em classes, podemos aproveitar a herança para reutilizar parte do código que temos em novos descritores. É isso que faremos na próxima seção.

Fábrica de propriedades versus classe descritora

Não é difícil implementar novamente a classe descritora melhorada do exemplo 20.2 acrescentando algumas linhas à fábrica de propriedades exibida no exemplo 19.24. A variável `__counter` é uma dificuldade, mas podemos deixá-la persistente entre as chamadas da fábrica definindo-a como um atributo do próprio objeto-função da fábrica, como mostra o exemplo 20.5.

Exemplo 20.5 – `bulkfood_v4prop.py`: mesma funcionalidade do exemplo 20.2, com uma fábrica de propriedades em vez de uma classe descritora

```
def quantity(): ❶
    try:
        quantity.counter += 1 ❷
    except AttributeError:
        quantity.counter = 0 ❸
    storage_name = '_{}:{}'.format('quantity', quantity.counter) ❹
    def qty_getter(instance): ❺
        return getattr(instance, storage_name)
    def qty_setter(instance, value):
        if value > 0:
            setattr(instance, storage_name, value)
        else:
            raise ValueError('value must be > 0')
    return property(qty_getter, qty_setter)
```

- ❶ Não precisamos mais do argumento `storage_name`.
- ❷ Não podemos contar com atributos de classe para compartilhar `counter` entre chamadas, portanto ele é definido como um atributo da própria função `quantity`.
- ❸ Se `quantity.counter` não está definido, atribuí-lhe o valor 0.
- ❹ Também não temos atributos de instância, portanto criamos `storage_name` como uma variável local e contamos com closures para mantê-la viva e ser usada depois por `qty_getter` e `qty_setter`.
- ❺ O restante do código é idêntico ao exemplo 19.24, mas, nesse caso, podemos usar as funções embutidas `getattr` e `setattr` em vez de lidar com `instance.__dict__`.

Então qual das opções você prefere? O exemplo 20.2 ou o exemplo 20.5?

Prefiro a abordagem com a classe descritora principalmente por dois motivos:

- Uma classe descritora pode ser estendida por herança; reutilizar código de uma função de fábrica sem copiar e colar é muito mais difícil.
- É mais legível armazenar estados em atributos de classe e de instância que em atributos de função e em closures, como precisamos fazer no exemplo 20.5.

Por outro lado, quando explico o exemplo 20.5, não sinto a necessidade de desenhar engenhocas e bugigangas. O código da fábrica de propriedades não depende de um relacionamento estranho entre instâncias e classes, evidenciado pelos argumentos `self`, `instance` e `owner`, passados para os métodos.

Resumindo, o padrão de fábrica de propriedades é mais simples em alguns aspectos, mas a abordagem com classe descritora é mais extensível. Também é a técnica mais amplamente usada.

Linitem tomada #5: um novo tipo descritor

A loja fictícia de produtos orgânicos atingiu um obstáculo: de alguma forma, uma instância de um item de um pedido foi criada com uma descrição em branco e o pedido não pode ser atendido. Para evitar isso, criaremos um novo descritor, `NonBlank`. Ao projetar `NonBlank`, percebemos que ele seria muito parecido com o descritor `Quantity`, exceto pela lógica de validação.

Refletindo sobre a funcionalidade de `Quantity`, notamos que ele faz duas tarefas distintas: cuida dos atributos de armazenagem nas instâncias gerenciadas e valida o valor usado para atualizar esses atributos. Isso pede uma refatoração, produzindo duas classes-bases:

AutoStorage

Classe de descritor que gerencia atributos de armazenagem automaticamente.

Validated

Subclasse abstrata de `AutoStorage` que sobrescreve o método `__set__`, chamando um método `validate` que deve ser implementado pelas subclasses.

Então reescreveremos `Quantity` e implementaremos `NonBlank` herdando de `Validated` e simplesmente escrevendo os métodos `validate`. A figura 20.5 representa essa configuração.

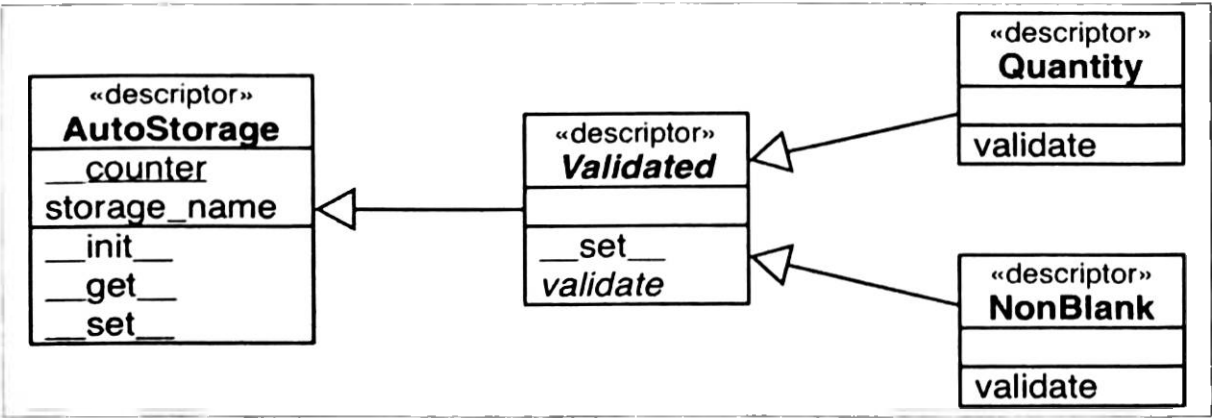


Figura 20.5 – Uma hierarquia de classes descritoras. A classe-base `AutoStorage` gerencia a armazenagem automática do atributo, `Validated` trata a validação delegando a um método abstrato `validate`, `Quantity` e `NonBlank` são subclasses concretas de `Validated`.

A relação entre `Validated`, `Quantity` e `NonBlank` é uma aplicação do padrão de projeto Template Method (Método Template). Em particular, `Validated.__set__` é um exemplo claro do que a Gangue dos Quatro descreve como método template:

Um método template define um algoritmo em termos de operações abstratas que as subclasses sobreescrevem para oferecer comportamentos concretos.⁴

Nesse caso, a operação abstrata é a validação. O exemplo 20.6 mostra a implementação das classes da figura 20.5.

Exemplo 20.6 – `model_v5.py`: as classes de descritor refatoradas

```
import abc

class AutoStorage: ❶
    __counter = 0
    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '{}#{{}'.format(prefix, index)
        cls.__counter += 1
    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return getattr(instance, self.storage_name)
    def __set__(self, instance, value):
        setattr(instance, self.storage_name, value) ❷

class Validated(abc.ABC, AutoStorage): ❸
    def __set__(self, instance, value):
        value = self.validate(instance, value) ❹
        super().__set__(instance, value) ❺
    @abc.abstractmethod
    def validate(self, instance, value): ❻
        """devolve valor validado ou levanta ValueError"""

class Quantity(Validated): ❼
    """um número maior que zero"""
```

⁴ Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 326.

```
def validate(self, instance, value):
    if value <= 0:
        raise ValueError('value must be > 0')
    return value
```

```
class NonBlank(Validated):
    """uma string com pelo menos um caractere não-branco"""
    def validate(self, instance, value):
        value = value.strip()
        if len(value) == 0:
            raise ValueError('value cannot be empty or blank')
        return value ❸
```

- ❶ AutoStorage oferece a maior parte das funcionalidades do descritor `Quantity` anterior...
- ❷ ... exceto validação.
- ❸ `Validated` é abstrata, mas também herda de `AutoStorage`.
- ❹ `__set__` delega a validação a um método `validate`...
- ❺ ... e então usa o valor devolvido para chamar `__set__` em uma superclasse, que executa a armazenagem propriamente dita.
- ❻ Nessa classe, `validate` é um método abstrato.
- ❼ `Quantity` e `NonBlank` herdam de `Validated`.
- ❽ Exigir que os métodos `validate` concretos devolvam o `value` validado lhes dá uma oportunidade de limpar, converter ou normalizar os dados recebidos. Nesse caso, `value` é devolvido sem os caracteres em branco no início e no fim.

Usuários de `model_v5.py` não precisam saber de todos esses detalhes. O que importa é que eles possam usar `Quantity` e `NonBlank` para automatizar a validação dos atributos de instância. Veja a classe `LineItem` mais recente no exemplo 20.7.

Exemplo 20.7 – `bulkfood_v5.py`: `LineItem` usando descritores `Quantity` e `NonBlank`

```
import model_v5 as model ❶
```

```
class LineItem:
    description = model.NonBlank() ❷
    weight = model.Quantity()
    price = model.Quantity()
    def __init__(self, description, weight, price):
        self.description = description
```

```

    self.weight = weight
    self.price = price

    def subtotal(self):
        return self.weight * self.price

```

- ❶ Importa o módulo `model_v5` dando-lhe um nome mais amigável.
- ❷ Coloca `model.NonBlank` em uso. O restante do código não foi alterado.

Os exemplos de `LineItem` que vimos neste capítulo mostram um uso típico de descritores para gerenciar atributos de dados. Um descritor desse tipo também é chamado de descritor dominante (*overriding descriptor*) porque seu método `__set__` domina o comportamento usual de escrita em um atributo de mesmo nome na instância gerenciada. No entanto há também descritores não dominantes (*nonoverriding descriptors*). Exploraremos essa distinção em detalhes na próxima seção.

Descritores dominantes e não dominantes

Lembre-se de que há uma assimetria importante no modo como Python trata atributos. Ler um atributo por meio de uma instância normalmente devolve o atributo definido na instância, mas, se um atributo desse tipo não existir na instância, um atributo da classe será recuperado. Por outro lado, escrever um valor em um atributo de instância cria (ou altera) o atributo na instância sem afetar a classe.

Essa assimetria também afeta os descritores, criando, na prática, duas categorias amplas de descritores, conforme o método `__set__` esteja ou não definido. Observar os diferentes comportamentos exige algumas classes, portanto usaremos o código do exemplo 20.8 como nossa base de testes nas próximas seções.



Todos os métodos `__get__` e `__set__` do exemplo 20.8 chamam `print_args` para que suas invocações sejam exibidas de forma legível. Entender `print_args` e as funções auxiliares `cls_name` e `display` não é importante, portanto não desvie sua atenção por causa delas.

Exemplo 20.8 – `descriptorkinds.py`: classes simples para estudar os comportamentos de descritores dominantes e não dominantes

```

### funções auxiliares somente para exibição ###

def cls_name(obj_or_cls):
    cls = type(obj_or_cls)
    if cls is type:
        cls = obj_or_cls
    return cls.__name__.split('.')[1]

```

```

def display(obj):
    cls = type(obj)
    if cls is type:
        return '<class {}>'.format(obj.__name__)
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return '<{} object>'.format(cls_name(obj))

def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
    print('-> {}.__{}__({})'.format(cls_name(args[0]), name, pseudo_args))

### classes essenciais nesse exemplo ###

class Overriding: # descritor dominante ❶
    """também conhecido como descritor de dados (data descriptor) ou enforced
    descriptor (descritor impositivo)"""
    def __get__(self, instance, owner):
        print_args('get', self, instance, owner) ❷
    def __set__(self, instance, value):
        print_args('set', self, instance, value)

class OverridingNoGet: # descritor dominante ❸
    """um descritor dominante sem ``__get__``"""
    def __set__(self, instance, value):
        print_args('set', self, instance, value)

class NonOverriding: # descritor não-dominante ❹
    """também conhecido como descritor não-dado (non-data descriptor) ou shadowable
    descriptor (descritor mascarável)"""
    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)

class Managed: ❺
    over = Overriding()
    over_no_get = OverridingNoGet()
    non_over = NonOverriding()
    def spam(self): ❻
        print('-> Managed.spam({})'.format(display(self)))

```

❶ Uma classe típica de descritor dominante, com `__get__` e `__set__`.

- ❷ A função `print_args` é chamada por todo método de descritor nesse exemplo.
- ❸ Um descritor dominante sem método `__get__`.
- ❹ Não há método `__set__` aqui, portanto esse é um descritor não dominante.
- ❺ A classe gerenciada usando uma instância de cada uma das classes descritoras.
- ❻ O método `spam` está aqui para comparação porque métodos também são descritores.

Nas seções a seguir, examinaremos o comportamento de leituras e escritas em atributos na classe `Managed` e em uma instância dela, analisando cada um dos diferentes descritores definidos.

Descritor dominante

Um descritor que implementa o método `__set__` é chamado de *descritor dominante* (overriding descriptor) porque, embora seja um atributo de classe, um descritor que implementa `__set__` interceptará tentativas de escrever no atributo de instância gerenciado por ele. É assim que o exemplo 20.2 foi implementado. Propriedades também são descritores dominantes: se você não fornecer uma função setter, o `__set__` default da classe `property` levantará `AttributeError` para informar que o atributo é somente de leitura. Dado o código do exemplo 20.8, experimentos com o descritor dominante podem ser vistos no exemplo 20.9.

Exemplo 20.9 – Comportamento de um descritor dominante: `obj.over` é uma instância de `Overriding` (Exemplo 20.8)

```
>>> obj = Managed() ❶
>>> obj.over ❷
-> Overriding.__get__(<Overriding object>, <Managed object>,
    <class Managed>)
>>> Managed.over ❸
-> Overriding.__get__(<Overriding object>, None, <class Managed>)
>>> obj.over = 7 ❹
-> Overriding.__set__(<Overriding object>, <Managed object>, 7)
>>> obj.over ❺
-> Overriding.__get__(<Overriding object>, <Managed object>,
    <class Managed>)
>>> obj.__dict__['over'] = 8 ❻
>>> vars(obj) ❼
{'over': 8}
>>> obj.over ❻
-> Overriding.__get__(<Overriding object>, <Managed object>,
    <class Managed>)
```

- ❶ Cria um objeto `Managed` para testes.
- ❷ `obj.over` dispara o método `__get__` do descritor, passando a instância gerenciada `obj` como segundo argumento.
- ❸ `Managed.over` dispara o método `__get__` do descritor, passando `None` como segundo argumento (`instance`).
- ❹ Atribuir valor a `obj.over` dispara o método `__set__` do descritor, passando o valor 7 como último argumento.
- ❺ Ler `obj.over` continua chamando o método `__get__` do descritor.
- ❻ Ignorando o descritor, definindo um valor diretamente em `obj.__dict__`.
- ❼ Confere se o valor está em `obj.__dict__`, com a chave `over`.
- ❽ No entanto, mesmo com um atributo de instância chamado `over`, o descritor `Managed.over` continua interceptando tentativas de ler `obj.over`.

Descritor dominante sem `__get__`

Normalmente, descritores dominantes implementam tanto `__set__` quanto `__get__`, mas também é possível implementar apenas `__set__`, como vimos no exemplo 20.1. Nesse caso, somente a escrita é tratada pelo descritor. Ler o descritor por meio de uma instância devolverá o próprio objeto descritor, pois não há `__get__` para tratar esse acesso. Mas se um atributo de instância de mesmo nome for criado colocando um valor diretamente no `__dict__` da instância, a leitura desse atributo simplesmente devolverá o valor guardado no `__dict__` da instância em vez de devolver o objeto descritor, que é um atributo da classe. Enquanto isso, o método `__set__` continuará interceptando novas tentativas de escrever nesse atributo de instância. Em outras palavras, um atributo na instância pode encobrir o descritor, mas somente na leitura. Veja o exemplo 20.10.

Exemplo 20.10 – Descritor dominante sem `__get__`: `obj.over_no_get` é uma instância de `OverridingNoGet` (Exemplo 20.8)

```
>>> obj.over_no_get ❶
<__main__.OverridingNoGet object at 0x665bcc>
>>> Managed.over_no_get ❷
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.over_no_get = 7 ❸
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get ❹
<__main__.OverridingNoGet object at 0x665bcc>
```

```
>>> obj.__dict__['over_no_get'] = 9 ❸
>>> obj.over_no_get ❹
9
>>> obj.over_no_get = 7 ❺
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get ❻
9
```

- ❶ Esse descritor dominante não tem um método `__get__`, portanto ler `obj.over_no_get` recupera a instância do descritor que está na classe.
- ❷ O mesmo ocorre se recuperarmos a instância do descritor diretamente da classe gerenciada.
- ❸ Tentar definir um valor em `obj.over_no_get` chama o método `__set__` do descritor.
- ❹ Como nosso `__set__` não faz alterações, ler `obj.over_no_get` novamente recupera a instância do descritor que está na classe gerenciada.
- ❺ Usando o `__dict__` da instância para escrever em um atributo da instância chamado `over_no_get`.
- ❻ Agora esse atributo de instância `over_no_get` encobre o descritor, mas somente para leitura.
- ❼ Tentar atribuir um valor a `obj.over_no_get` continua passando pelo set do descritor.
- ❽ Mas, na leitura, esse descritor ser encoberto enquanto houver um atributo de instância de mesmo nome.

Descritor não dominante

Se um descritor não implementa `__set__`, ele é um descritor não dominante (*nonoverriding descriptor*). Escrever em um atributo de instância de mesmo nome encobrirá o descritor, deixando-o sem efeito para tratar esse atributo nessa instância específica. Métodos são implementados como descritores não dominantes. O exemplo 20.11 mostra o funcionamento de um descritor não dominante.

Exemplo 20.11 – Comportamento de um descritor não dominante; `obj.non_over` é uma instância de `NonOverriding` (Exemplo 20.8)

```
>>> obj = Managed()
>>> obj.non_over ❶
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>,
    <class Managed>)
>>> obj.non_over = 7 ❷
```

```
>>> obj.non_over ❸
7
>>> Managed.non_over ❹
-> NonOverriding.__get__(<NonOverriding object>, None, <class Managed>)
>>> del obj.non_over ❺
>>> obj.non_over ❻
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>,
    <class Managed>)
```

- ❶ `obj.non_over` dispara o método `__get__` do descritor, passando `obj` como segundo argumento.
- ❷ `Managed.non_over` é um descritor não dominante, portanto não há `__set__` para interferir nessa atribuição.
- ❸ `obj` agora tem um atributo de instância chamado `non_over`, que encobre o atributo de descritor de mesmo nome na classe `Managed`.
- ❹ O descritor `Managed.non_over` continua presente e captura esse acesso por meio da classe.
- ❺ Se o atributo de instância `non_over` é apagado...
- ❻ ... ler `obj.non_over` acessa o método `__get__` do descritor na classe, mas observe que o segundo argumento é a instância gerenciada.



Colaboradores de Python e autores que escrevem sobre essa linguagem usam termos diferentes quando discutem esses conceitos. Descritores dominantes são também chamados de descritores de dados (data descriptors) ou enforced descriptors (descritores impositivos). Descritores não dominantes são também conhecidos como descritores não dados (nondata descriptors) ou shadowable descriptors (descritores toldáveis).

Nos exemplos anteriores, vimos várias escritas em um atributo de instância de mesmo nome que um descritor, e diferentes resultados de acordo com a presença ou a ausência de um método `__set__` no descritor.

A escrita de atributos na classe não pode ser controlada por descritores associados à mesma classe. Em particular, isso quer dizer que os atributos da classe que são descritores podem ser sobrescritos por atribuições na própria classe, como explica a próxima seção.

Sobrescrevendo um descritor na classe

Independentemente de um descritor ser dominante ou não dominante, ele pode ser sobrescrito por meio de atribuição à classe. É uma técnica de monkey-patching, mas, no exemplo 20.12, os descritores são substituídos por inteiros, o que efetivamente quebraria qualquer classe que dependa dos descritores para um funcionamento apropriado.

Exemplo 20.12 – Qualquer descritor pode ser sobrescrito na própria classe

```
>>> obj = Managed() ❶
>>> Managed.over = 1 ❷
>>> Managed.over_no_get = 2
>>> Managed.non_over = 3
>>> obj.over, obj.over_no_get, obj.non_over ❸
(1, 2, 3)
```

- ❶ Cria uma nova instância para testar depois.
- ❷ Sobrescreve os atributos de descritor na classe.
- ❸ Os descritores realmente se foram.

O exemplo 20.12 mostra outra assimetria da leitura e da escrita de atributos: embora a leitura de um atributo de classe possa ser controlada por um descritor com `__get__` associado à classe gerenciada, a escrita de um atributo de classe não pode ser tratada por um descritor com `__set__` associado à mesma classe.



Para controlar a escrita de atributos em uma classe, você precisa associar descritores à classe da classe – em outras palavras, à metaclasses. Por padrão, a metaclasses de classes definidas pelo usuário é `type`, e você não pode adicionar atributos a `type`. Mas, no capítulo 21, criaremos nossas próprias metaclasses.

Vamos nos concentrar agora em como os descritores são usados para implementar métodos em Python.

Métodos são descritores

Uma função em uma classe torna-se um método vinculado (bound method) porque todas as funções definidas pelo usuário têm um método `__get__`; sendo assim, elas funcionam como descritores quando associadas a uma classe. O exemplo 20.13 mostra a leitura do método `spam` da classe `Managed` apresentada no exemplo 20.8.

Exemplo 20.13 – Um método é um descritor não dominante

```
>>> obj = Managed()
>>> obj.spam ❶
<bound method Managed.spam of <descriptor kinds.Managed object at 0x74c80c>>
>>> Managed.spam ❷
<function Managed.spam at 0x734734>
>>> obj.spam = 7 ❸
>>> obj.spam
7
```

- ❶ Ler de `obj.spam` recupera um objeto de método vinculado a uma instância (bound method).
- ❷ Mas ler de `Managed.spam` recupera uma função.
- ❸ Atribuir um valor a `obj.spam` encobre o atributo da classe, deixando o método `spam` inacessível a partir da instância `obj`.

Como funções não implementam `__set__`, elas são descritores não dominantes, como a última linha do exemplo 20.13 mostra.

A outra lição fundamental do exemplo 20.13 é que `obj.spam` e `Managed.spam` recuperam objetos diferentes. Como ocorre normalmente com descritores, o `__get__` de uma função devolve uma referência a si mesmo quando o acesso ocorre por meio da classe gerenciada. Porém, quando o acesso é feito por meio de uma instância, o `__get__` da função devolve um objeto que é o método vinculado: um invocável que encapsula a função e vincula a instância gerenciada (por exemplo, `obj`) ao primeiro argumento da função (isto é, `self`), como faz a função `functools.partial` (conforme vimos na seção “Congelando argumentos com `functools.partial`” na página 198).

Para entender melhor esse mecanismo, dê uma olhada no exemplo 20.14.

Exemplo 20.14 – `method_is_descriptor.py`: uma classe `Text`, derivada de `UserString`

```
import collections

class Text(collections.UserString):
    def __repr__(self):
        return 'Text({!r})'.format(self.data)
    def reverse(self):
        return self[::-1]
```

Agora vamos investigar o método `Text.reverse`. Veja o exemplo 20.15.

Exemplo 20.15 – Experimentos com um método

```

>>> word = Text('forward')
>>> word ❶
Text('forward')
>>> word.reverse() ❷
Text('drawof')
>>> Text.reverse(Text('backward')) ❸
Text('drawkcab')
>>> type(Text.reverse), type(word.reverse) ❹
(<class 'function'>, <class 'method'>)
>>> list(map(Text.reverse, ['repaid', (10, 20, 30), Text('stressed')])) ❺
['diaper', (30, 20, 10), Text('desserts')]
>>> Text.reverse.__get__(word) ❻
<bound method Text.reverse of Text('forward')>
>>> Text.reverse.__get__(None, Text) ❼
<function Text.reverse at 0x101244e18>
>>> word.reverse ❽
<bound method Text.reverse of Text('forward')>
>>> word.reverse.__self__ ❾
Text('forward')
>>> word.reverse.__func__ is Text.reverse ❿
True

```

- ❶ A repr de uma instância de Text é semelhante a um chamado ao construtor Text que criaria uma instância igual.
- ❷ O método reverse devolve o texto invertido.
- ❸ Um método chamado na classe funciona como uma função.
- ❹ Observe os tipos diferentes: uma function e um method.
- ❺ Text.reverse opera como uma função, lidando até mesmo com objetos que não são instâncias de Text.
- ❻ Toda função é um descritor não dominante. Chamar seu __get__ com uma instância recupera um método vinculado a essa instância.
- ❼ Chamar o __get__ da função com None como o argumento instance recupera a própria função.
- ❽ A expressão word.reverse na verdade chama Text.reverse.__get__(word), devolvendo o método vinculado.
- ❾ O objeto que representa o método vinculado tem um atributo __self__ que armazena uma referência à instância em que o método foi chamado.
- ❿

- ⑩ O atributo `__func__` do método vinculado é uma referência à função original associada à classe gerenciada.

O objeto do método vinculado também tem um método `__call__` que trata a chamada propriamente dita. Esse método chama a função original referenciada em `__func__`, passando o atributo `__self__` do método como primeiro argumento. É assim que a vinculação implícita do argumento convencional `self` funciona.

A transformação de funções em métodos vinculados é um excelente exemplo de como descritores são usados como infraestrutura na linguagem.

Após esse profundo mergulho no funcionamento de descritores e métodos, vamos ver alguns conselhos práticos sobre seu uso.

Dicas para uso de descritores

A lista a seguir aborda algumas consequências práticas das características dos descritores que acabamos de apresentar:

Use `property` para Manter a Simplicidade (Keep It Simple)

A classe embutida `property` na verdade cria descritores dominantes implementando tanto `__set__` quanto `__get__`, mesmo que você não defina um método setter. O `__set__` default de uma propriedade levanta `AttributeError: can't set attribute` (`AttributeError: atributo não pode receber valor`), portanto uma propriedade é a maneira mais fácil de criar um atributo somente de leitura, evitando o problema descrito a seguir.

Descritores somente de leitura exigem `__set__`

Se você usar uma classe descritora para implementar um atributo somente de leitura, lembre-se de implementar tanto `__get__` quanto `__set__`; caso contrário, escrever em um atributo de mesmo nome em uma instância encobrirá o descritor. O método `__set__` de um atributo somente de leitura deve apenas levantar `AttributeError` com uma mensagem adequada.⁵

Descritores de validação podem funcionar somente com `__set__`

Em um descritor projetado somente para validação, o método `__set__` deve verificar o argumento `value` recebido e, se for válido, deve escrevê-lo diretamente no `__dict__`

5 Python não é consistente nessas mensagens. Tentar mudar o atributo `c.real` de um número do tipo `complex` resulta em `AttributeError: read-only attribute` (`AttributeError: atributo somente de leitura`), mas uma tentativa de alterar `c.conjugate` (um método de `complex`) resulta em `AttributeError: 'complex' object attribute 'conjugate' is read-only` (`AttributeError: atributo 'conjugate' do objeto 'complex' somente de leitura`).

da instância usando o nome da instância do descritor como chave. Dessa maneira, ler o atributo de mesmo nome na instância será tão rápido quanto possível, pois não exige um `__get__`. Veja o código no exemplo 20.1.

Caching pode ser feito de modo eficiente somente com `__get__`

Se você implementar somente o método `__get__`, terá um descritor não dominante. Eles são úteis para realizar algum processamento custoso e cachear o resultado escrevendo em um atributo de mesmo nome na instância. O atributo de instância de mesmo nome encobrirá o descritor, de modo que acessos subsequentes a esse atributo o buscarão diretamente no `__dict__` da instância, sem disparar o `__get__` do descritor.

Métodos não especiais podem ser encobertos por atributos de instância

Como funções e métodos implementam apenas `__get__`, eles não tratam tentativas de escrever em atributos de instância de mesmo nome, portanto uma atribuição simples como `my_obj.the_method = 7` significa que acessos subsequentes a `the_method` por meio dessa instância recuperarão o número 7 – sem afetar a classe ou outras instâncias. No entanto esse problema não interfere em métodos especiais. O interpretador só procura métodos especiais na própria classe; em outras palavras, `repr(x)` é executado como `x.__class__.__repr__(x)`, de modo que um atributo `__repr__` definido em `x` não tem nenhum efeito em `repr(x)`. Pelo mesmo motivo, a existência de um atributo chamado `__getattr__` em uma instância não subverterá o algoritmo usual de acesso a atributos.

O fato de métodos não especiais poderem ser sobrescritos tão facilmente em instâncias pode soar como algo frágil e suscetível a erros, mas, pessoalmente, nunca tive problemas com isso em mais de quinze anos escrevendo código Python. Por outro lado, se você faz muita criação dinâmica de atributos, em que os nomes dos atributos são provenientes de dados que não estão sob seu controle (como fizemos no início deste capítulo), você deve estar ciente disso e, talvez, implementar filtros ou escapar os nomes dos atributos dinâmicos para preservar sua sanidade.



A classe `FrozenJSON` do exemplo 19.6 está protegida contra atributos de instância encobrindo métodos, pois seus únicos métodos são métodos especiais e o método de classe `build`. Métodos de classe estão protegidos, desde que sejam sempre acessados por meio da classe, como fiz com `FrozenJSON.build` no exemplo 19.6 – substituído depois por `__new__` no exemplo 19.7. A classe `Record` (Exemplos 19.9 e 19.11) e as subclasses também estão protegidas: elas só usam métodos especiais, métodos de classe, métodos estáticos e propriedades. Propriedades são descritores dominantes, portanto não podem ser sobrescritas por atributos de instância.

Para encerrar este capítulo, discutiremos dois recursos que vimos com propriedades, mas não abordamos no contexto de descritores: documentação e tratamento de tentativas de apagar um atributo gerenciado.

Docstring de descritores e controle de remoção

A docstring de uma classe descritora é usada para documentar todas as instâncias do descritor na classe gerenciada. Veja a figura 20.6, que mostra as ajudas exibidas para a classe `LineItem` com os descritores `Quantity` e `NonBlank` dos exemplos 20.6 e 20.7.

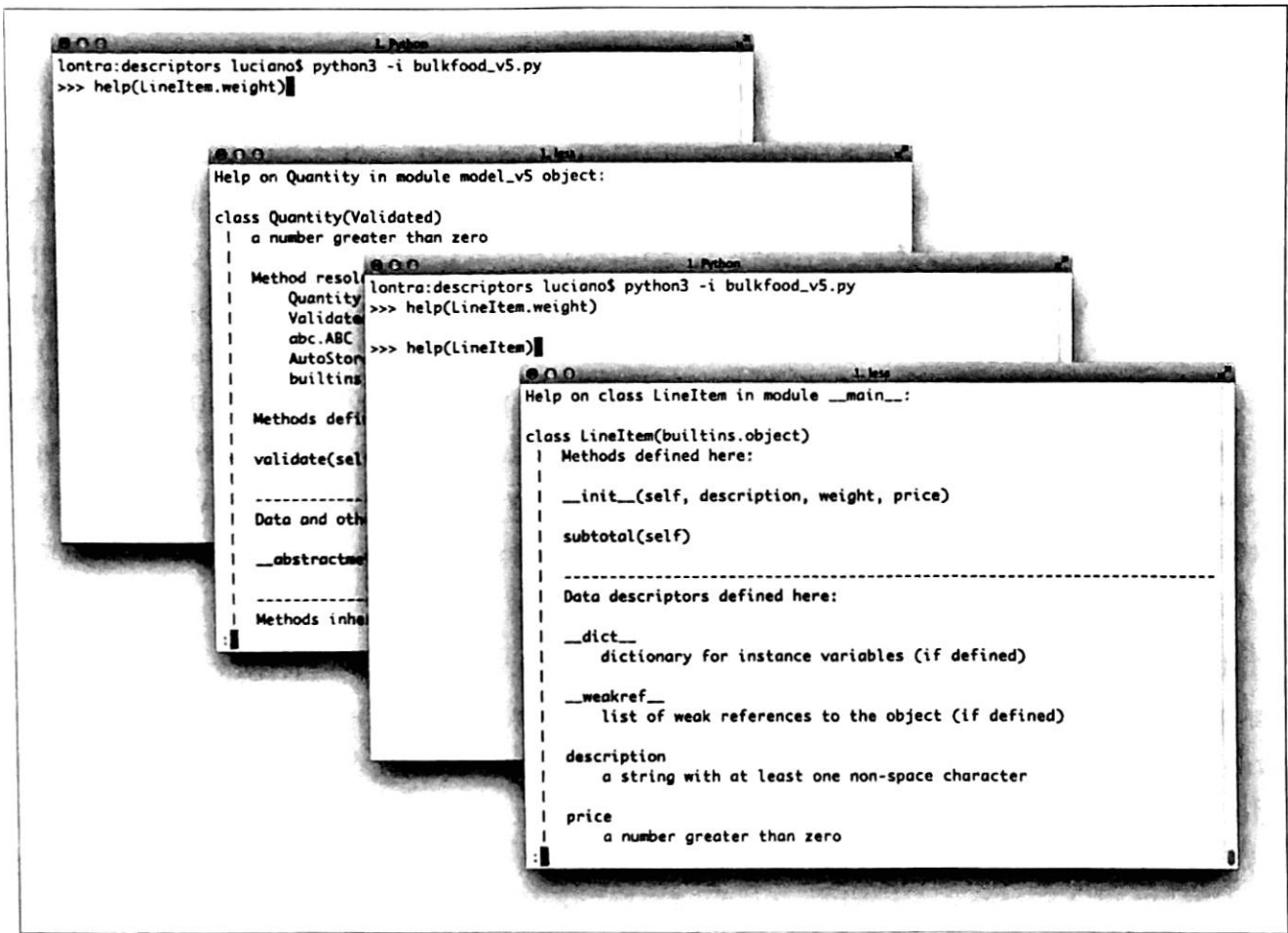


Figura 20.6 – Capturas de tela do console de Python com a execução dos comandos `help(LineItem.weight)` e `help(LineItem)`.

De certo modo, isso é insatisfatório. No caso de `LineItem`, seria bom acrescentar, por exemplo, a informação de que `weight` deve estar em quilogramas. Isso seria trivial com propriedades, pois cada propriedade trata um atributo gerenciado específico. Mas com descritores a mesma classe descritora `Quantity` é usada para `weight` e `price`.⁶

O segundo detalhe que discutimos com as propriedades, mas não abordamos com descritores, é o tratamento de tentativas de apagar um atributo gerenciado. Isso pode

⁶ Personalizar o texto de ajuda para cada instância de descritor é surpreendentemente difícil. Uma solução exige criar dinamicamente uma classe que encapsule cada instância do descritor.

ser feito implementando um método `__delete__` na classe descritora. Implementar uma classe tola de descritor com `__delete__` fica como exercício para o leitor sem pressa.

Resumo do capítulo

O primeiro exemplo deste capítulo foi uma continuação dos exemplos de `LineItem` do capítulo 19. No exemplo 20.1, substituímos propriedades por descritores. Vimos que um descritor é uma classe cujas instâncias são instaladas como atributos da classe gerenciada. A discussão desse mecanismo exigiu uma terminologia especial, introduzindo termos como instância gerenciada e atributo de armazenagem.

Na seção “`LineItem` tomada #4: nomes automáticos para atributos de armazenagem” na página 693, eliminamos o requisito de declarar descritores `Quantity` com um `storage_name` explícito, o que era redundante e suscetível a erros, pois esse nome deveria sempre coincidir com o nome do atributo à esquerda da atribuição na instanciação do descritor. A solução foi gerar valores únicos para `storage_name`, combinando o nome da classe descritora com um contador no nível da classe (por exemplo, `'_Quantity#1'`).

Em seguida, comparamos o tamanho do código e os pontos fortes e fracos de uma classe descritora com uma fábrica de propriedades criada com base em técnicas de programação funcional. A fábrica de propriedades funciona perfeitamente e é mais simples em alguns aspectos, mas a classe descritora é mais flexível e é a solução padrão. Uma vantagem essencial da classe descritora foi explorada na seção “`LineItem` tomada #5: um novo tipo descrito” na página 700: usar subclasses para compartilhar código criando descritores especializados aproveitando funcionalidades comuns.

Em seguida, vimos os diferentes comportamentos dos descritores que oferecem ou omitem o método `__set__`, fazendo a distinção crucial entre descritores dominantes e não-dominantes. Por meio de testes detalhados, mostramos quando os descritores estão no controle e quando são encobertos, ignorados ou sobrescritos.

Depois disso, estudamos uma categoria particular de descritores não-dominantes: os métodos. Testes de console mostraram como uma função associada a uma classe torna-se um método quando acessada por meio de uma instância, aproveitando o protocolo de descritores.

Para concluir o capítulo, a seção “Dicas para uso de descritores” na página 712 apresentou rapidamente como funcionam a remoção e a documentação de descritores.

Neste capítulo, vimos alguns problemas que podem ser resolvidos somente com metaprogramação de classes e os deixamos para o capítulo 21.

Leituras complementares

Além da referência obrigatória ao capítulo “Data Model” (Modelo de dados, <http://bit.ly/1GsZwss>), o *Descriptor HowTo Guide* (HowTo para descritores, <http://bit.ly/1HGwlS3>) de Raymond Hettinger é um material importante – parte da coleção de HowTos (<http://bit.ly/1HGwnsV>) da documentação oficial de Python.

Como de costume, em assuntos ligados ao modelo de objetos de Python, o livro *Python in a Nutshell*, 2E (O’Reilly) de Alex Martelli é altamente confiável e objetivo, apesar não ser uma obra tão recente: os mecanismos principais discutidos neste capítulo foram introduzidos em Python 2.2, muito antes da versão 2.5 considerada naquele livro. Martelli também tem uma apresentação cujo título é *Python’s Object Model* (Modelo de objetos de Python), que discute propriedades e descritores em detalhes (slides em <http://bit.ly/1HGwoxa>, vídeo em <http://bit.ly/1HGwp46>). Altamente recomendada.

Para Python 3 com exemplos práticos, *Python Cookbook*, 3E de David Beazley e Brian K. Jones (O’Reilly)⁷ tem várias receitas que ilustram descritores, entre as quais gostaria de destacar: “6.12. Reading Nested and Variable-Sized Binary Structures” (Ler estruturas binárias aninhadas e de tamanhos variáveis), “8.10. Using Lazily Computed Properties” (Usar propriedades processadas em modo lazy), “8.13. Implementing a Data Model or Type System” (Implementar um modelo de dados ou um sistema de tipos) e “9.9. Defining Decorators As Classes” (Definir decoradores como classes) – a última receita aborda questões profundas sobre a interação entre decoradores de função, descritores e métodos, explicando como um decorador de função implementado como uma classe com `__call__` também precisa implementar `__get__` se quiser funcionar com métodos de decoração e também com funções.

Ponto de vista

O problema com self

“Pior é Melhor” (Worse is Better) é uma filosofia de projeto descrita por Richard P. Gabriel em “The Rise of Worse is Better” (O surgimento de Pior é Melhor, <http://bit.ly/1HGwvlZ>). A primeira prioridade dessa filosofia é “Simplicidade”, sobre a qual Gabriel afirma o seguinte:

O design deve ser simples, tanto em implementação quanto em interface. É importante que a implementação seja simples, mais que a interface. Simplicidade é o aspecto mais importante a ser considerado em um projeto.

7 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

Acredito que o requisito de declarar explicitamente `self` como primeiro argumento em métodos seja uma aplicação de “Pior é Melhor” em Python. A implementação é simples – até mesmo elegante – à custa da interface de usuário: uma assinatura de método como `def zfill(self, width)`: não bate visualmente com a chamada a `pobox.zfill(8)`.

Modula-3 introduziu essa convenção – e o uso do identificador `self` –, mas há uma diferença: em Modula-3, as interfaces são declaradas separadamente de suas implementações e, na declaração da interface, o argumento `self` é omitido, de modo que, do ponto de vista do usuário, um método aparece em uma declaração de interface exatamente com o mesmo número de argumentos explícitos que ele recebe.

Uma melhoria quanto a isso tem sido as mensagens de erro: em um método definido pelo usuário com um argumento além de `self`, se o usuário chama `obj.meth()`, Python 2.7 levanta `TypeError: meth() takes exactly 2 arguments (1 given)` (`TypeError: meth() aceita exatamente dois argumentos (um dado)`), mas em Python 3.4 a mensagem é menos confusa, deixando de lado o problema do número de argumentos e nomeando o argumento ausente: `meth() missing 1 required positional argument: 'x'` (faltando um argumento posicional obrigatório em `meth()`: ‘x’).

Além do uso de `self` como argumento explícito, a necessidade de qualificar todo acesso a atributos de instância com `self` também é criticada.⁸ Pessoalmente, não me importo de digitar o qualificador `self`: ele é útil para distinguir variáveis locais de atributos. Meu problema é com o uso de `self` no comando `def`. Mas eu acabei me acostumando com ele, mesmo sem gostar.

Qualquer pessoa que não esteja satisfeita com o `self` explícito em Python poderá se sentir aliviada se considerar a surpreendente semântica do `this` implícito em JavaScript. Guido teve boas razões para fazer `self` funcionar como funciona e escreveu sobre isso em “Adding Support for User-Defined Classes” (Acréscendo suporte a classes definidas pelo usuário, <http://bit.ly/1CAyiQY>), um post em seu blog *The History of Python* (A história de Python).

⁸ Veja, por exemplo, o famoso post *Python Warts* (“Verrugas” de Python) de A. M. Kuchling (arquivado em <http://bit.ly/1cPSaDh>); o próprio Kuchling não se incomoda tanto com o qualificador `self`, mas o menciona – provavelmente dando eco a opiniões em `comp.lang.python`.

CAPÍTULO 21

Metaprogramação com classes

[Metaclasses] são mágicas tão profundas que 99% dos usuários jamais deveriam se preocupar com elas. Se você está se perguntando se precisa delas, é sinal de que não precisa (as pessoas que realmente precisam sabem, com certeza, que precisam delas, e não precisam de uma explicação para saber por quê).¹

— Tim Peters

Criador do algoritmo timsort e colaborador produtivo de Python

Metaprogramação com classes é a arte de criar ou customizar classes em tempo de execução. Classes são objetos de primeira classe em Python, portanto uma função pode ser usada para criar uma nova classe a qualquer momento sem usar a palavra reservada `class`. Decoradores de classe também são funções e podem inspecionar, alterar e até mesmo substituir a classe decorada por outra classe. Por fim, metaclasses são a ferramenta mais avançada para metaprogramação de classes: elas permitem criar categorias totalmente novas de classes com características especiais, por exemplo, as ABC (classes-base abstratas) que já vimos anteriormente.

Metaclasses são poderosas, mas difíceis de implementar corretamente. Decoradores de classe resolvem muitos dos mesmos problemas de modo mais simples. De fato, metaclasses atualmente são tão difíceis de justificar em casos práticos que meu exemplo didático favorito perdeu muito de seu atrativo com a introdução de decoradores de classe em Python 2.6.

Também discutiremos a distinção entre tempo de importação (import time) e tempo de execução (runtime): um pré-requisito essencial para metaprogramação eficaz em Python.

¹ Mensagem para `comp.lang.python`, assunto: “Acrimony in c.l.p.” (Animosidade em c.l.p., <http://bit.ly/1e8iABS>). É outra parte da mesma mensagem de 23 de dezembro de 2002, citada no prefácio. O TimBot estava inspirado naquele dia.



Esse é um assunto empolgante, e é fácil se deixar levar por ele. Portanto início este capítulo com a seguinte advertência: Se você não está construindo um framework, não escreva metaclasses – a menos que esteja fazendo isso por diversão ou para exercitar os conceitos.

Começaremos analisando a criação de uma classe em tempo de execução.

Uma fábrica de classes

A biblioteca-padrão tem uma fábrica de classes que vimos diversas vezes neste livro: `collections.namedtuple`. É uma função que, dado um nome de classe e considerando os nomes de atributos, cria uma subclasse de `tuple` que permite recuperar itens pelo nome e oferece uma boa `__repr__` para depuração.

Algumas vezes, senti necessidade de ter uma fábrica semelhante para objetos mutáveis. Suponha que eu esteja escrevendo uma aplicação para um pet shop e queira processar dados de cães como registros simples. É ruim ter um código repetitivo como este:

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

Maçante... O nome de cada campo aparece três vezes. Todo esse código repetitivo nem mesmo nos proporciona uma boa `repr`:

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

Inspirado em `collections.namedtuple`, vamos criar uma `record_factory` que crie classes simples como `Dog` sob demanda. O exemplo 21.1 mostra como isso funcionaria.

Exemplo 21.1 – Testando `record_factory`, uma fábrica simples de classe

```
>>> Dog = record_factory('Dog', 'name weight owner') ❶
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex ❷
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex ❸
>>> name, weight
('Rex', 30)
>>> "{2}'s dog weighs {1}kg".format(*rex) ❹
```

```
"Bob's dog weighs 30kg"
>>> rex.weight = 32 ❸
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
>>> Dog.__mro__ ❹
(<class 'factories.Dog'>, <class 'object'>)
```

- ❶ A assinatura da fábrica é semelhante à assinatura de `namedtuple`: nome da classe, seguido dos nomes dos atributos em uma única string, separados por espaços ou vírgulas.
- ❷ Uma boa repr.
- ❸ As instâncias são iteráveis, portanto podem ser convenientemente desempacotadas na atribuição...
- ❹ ... ou quando passadas para funções como `format`.
- ❺ Uma instância do registro é mutável.
- ❻ A classe recém-criada herda de `object` – não tem relação com nossa fábrica.

O código de `record_factory` está no exemplo 21.2.²

Exemplo 21.2 – `record_factory.py`: uma fábrica simples de classes

```
def record_factory(cls_name, field_names):
    try:
        field_names = field_names.replace(',', ' ').split() ❶
    except AttributeError: # sem .replace ou .split
        pass # supõe que já é uma sequência de identificadores
    field_names = tuple(field_names) ❷

    def __init__(self, *args, **kwargs): ❸
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self): ❹
        for name in self.__slots__:
            yield getattr(self, name)

    def __repr__(self): ❺
        values = ', '.join('{}={!r}'.format(*i) for i
                               in zip(self.__slots__, self))
        return '{}({})'.format(self.__class__.__name__, values)
```

² Agradeço ao meu amigo J. S. Bueno por ter sugerido essa solução.

```

cls_attrs = dict(__slots__ = field_names, ❶
                 __init__ = __init__,
                 __iter__ = __iter__,
                 __repr__ = __repr__)

return type(cls_name, (object,), cls_attrs) ❷

```

- ❶ Duck typing na prática: tenta separar `field_names` por vírgulas ou espaços; se isso falhar, supõe que já é um iterável, com um nome por item.
- ❷ Cria uma tupla de nomes de atributos que será o atributo `__slots__` da nova classe; também define a ordem dos campos para desempacotamento e `__repr__`.
- ❸ Essa função será o método `__init__` na nova classe. Aceita argumentos posicionais e/ou nomeados.
- ❹ Implementa um `__iter__` para que as instâncias da classe sejam iteráveis; produz os valores dos campos na ordem especificada por `__slots__`.
- ❺ Produz a repr elegante, iterando por `__slots__` e `self`.
- ❻ Monta o dicionário de atributos da classe.
- ❼ Cria e devolve a nova classe, chamando o construtor `type`.

Normalmente, pensamos em `type` como uma função, pois a usamos dessa forma, por exemplo, `type(my_object)` para obter a classe do objeto – é o mesmo que `my_object.__class__`. Mas `type` é uma classe. Ela se comporta como uma classe que cria uma nova classe quando chamada com três argumentos:

```

MyClass = type('MyClass', (MySuperClass, MyMixin),
               {'x': 42, 'x2': lambda self: self.x * 2})

```

Os três argumentos de `type` chamam-se `name`, `bases` e `dict` – o último é um mapeamento de nomes de atributos e atributos da nova classe. O código anterior é funcionalmente equivalente a:

```

class MyClass(MySuperClass, MyMixin):
    x = 42

    def x2(self):
        return self.x * 2

```

A novidade aqui é que as instâncias de `type` são classes, como `MyClass` nesse caso, ou a classe `Dog` no exemplo 21.1.

Em suma, a última linha de `record_factory` no exemplo 21.1 cria uma classe nomeada de acordo com o valor de `cls_name`, com `object` como sua única superclasse imediata e atributos de classe chamados `__slots__`, `__init__`, `__iter__` e `__repr__`, dos quais os últimos três são métodos de instância.

Poderíamos ter dado outro nome qualquer ao atributo de classe `__slots__`, mas então teríamos de implementar `__setattr__` para validar os nomes dos atributos recebendo valor, pois, em nossas classes para registros, queremos que o conjunto de atributos seja sempre o mesmo e esteja na mesma ordem. No entanto lembre-se de que a funcionalidade principal de `__slots__` é economizar memória quando lidamos com milhões de instâncias, e usar `__slots__` tem algumas desvantagens, discutidas na seção “Economizando espaço com o atributo de classe `__slots__`” na página 307.

Chamar `type` com três argumentos é um modo comum de criar uma classe dinamicamente. Se espiar o código-fonte de `collections.namedtuple` (<http://bit.ly/1HGwxRI>), você verá uma abordagem diferente: há um `_class_template`, que é um template de código-fonte em forma de string, e a função `namedtuple` preenche suas lacunas chamando `_class_template.format(...)`. A string de código-fonte resultante é então avaliada com a função embutida `exec`.



É uma boa prática evitar `exec` ou `eval` para metaprogramação em Python. Essas funções representam riscos sérios à segurança quando recebem strings (mesmo que sejam fragmentos) de fontes não confiáveis. Python oferece ferramentas suficientes para introspecção a ponto de `exec` e `eval` serem desnecessários na grande maioria das ocasiões. Entretanto os core developers de Python optaram por usar `exec` na implementação de `namedtuple`. A abordagem escolhida permite acessar o código-fonte da classe construída por meio do atributo `._source` (<http://bit.ly/1HGwAfW>).

Instâncias de classes criadas por `record_factory` têm uma limitação: elas não são serializáveis – ou seja, não podem ser usadas com as funções `dump/load` do módulo `pickle`. Resolver esse problema está além do escopo desse exemplo, que tem como objetivo mostrar a classe `type` em ação em um caso de uso simples. Para uma solução completa, estude o código-fonte de `collections.namedtuple` (<http://bit.ly/1HGwxRI>) e procure a palavra “pickling”.

Um decorador de classe para personalizar descritores

Quando deixamos o exemplo de `LineItem` na seção “`LineItem` tomada #5: um novo tipo de descritor” na página 700, o problema dos nomes de armazenagem descritivos continuava pendente: o valor de atributos como `weight` era armazenado em um atributo de instância chamado `_Quantity#0`, tornando a depuração um pouco complicada. Você pode recuperar o nome de armazenagem de um descritor do exemplo 20.7 com as linhas a seguir:

```
>>> LineItem.weight.storage_name
'_Quantity#0'
```

No entanto seria melhor se os nomes de armazenagem incluíssem o nome do atributo gerenciado, assim:

```
>>> LineItem.weight.storage_name  
'_Quantity#weight'
```

Lembre-se de que, de acordo com a seção “LineItem tomada #4: nomes automáticos para atributos de armazenagem” na página 693, não podíamos usar nomes de armazenagem descritivos porque quando o descritor é instanciado, ele não tem como saber o nome do atributo gerenciado (ou seja, o atributo de classe ao qual o descritor será vinculado, por exemplo, `weight`, nos exemplos anteriores). Mas depois que a classe toda é montada e os descritores são vinculados aos atributos da classe, podemos inspecionar a classe e definir nomes apropriados de armazenagem aos descritores. Isso poderia ser feito no método `__new__` da classe `LineItem`, de modo que, quando os descritores fossem usados no método `__init__`, os nomes de armazenagem corretos já estivessem definidos. O problema de usar `__new__` para isso é o processamento desperdiçado: a lógica de `__new__` será executada sempre que uma nova instância de `LineItem` for criada, mas a vinculação do descritor ao atributo gerenciado não mudará depois que a classe `LineItem` for criada. Portanto precisamos definir os nomes de armazenagem quando a classe é criada. Isso pode ser feito com um decorador de classe ou com uma metaclasses. Faremos primeiro do modo mais fácil.

Um decorador de classe é bem parecido com um decorador de função: é uma função que recebe um objeto-classe e devolve a mesma classe ou uma classe modificada.

No exemplo 21.3, a classe `LineItem` será avaliada pelo interpretador e o objeto classe resultante será passado para a função `model.entity`. Python vinculará o nome global `LineItem` com o que quer que a função `model.entity` devolver. Nesse exemplo, `model.entity` devolve a mesma classe `LineItem` com o atributo `storage_name` de cada instância de descritor alterada.

Exemplo 21.3 – `bulkfood_v6.py`: `LineItem` usando descritores `Quantity` e `NonBlank`

```
import model_v6 as model  
  
@model.entity ❶  
class LineItem:  
    description = model.NonBlank()  
    weight = model.Quantity()  
    price = model.Quantity()  
  
    def __init__(self, description, weight, price):  
        self.description = description  
        self.weight = weight  
        self.price = price
```

```
def subtotal(self):
    return self.weight * self.price
```

❶ A única mudança nessa classe é o decorador adicionado.

O exemplo 21.4 mostra a implementação do decorador. Somente o código novo no final de *model_v6.py* é mostrado aqui; o restante do módulo é idêntico a *model_v5.py* (Exemplo 20.6).

Exemplo 21.4 – *model_v6.py*: um decorador de classe

```
def entity(cls): ❶
    for key, attr in cls.__dict__.items(): ❷
        if isinstance(attr, Validated): ❸
            type_name = type(attr).__name__
            attr.storage_name = '_{}#{}'.format(type_name, key) ❹
    return cls ❺
```

- ❶ O decorador recebe a classe como argumento.
- ❷ Itera por dict, que armazena os atributos da classe.
- ❸ Se o atributo for um de nossos descritores Validated...
- ❹ ... define o `storage_name` para que use o nome da classe descritora e o nome do atributo gerenciado (por exemplo, `_NonBlank#description`).
- ❺ Devolve a classe modificada.

Os doctests em *bulkfood_v6.py* provam que as alterações foram bem-sucedidas. O exemplo 21.5 mostra os nomes dos atributos de armazenagem em uma instância de `LineItem`.

Exemplo 21.5 – *bulkfood_v6.py*: doctests para os novos atributos `storage_name` dos descritores

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> dir(raisins)[:3]
['_NonBlank#description', '_Quantity#price', '_Quantity#weight']
>>> LineItem.description.storage_name
'_NonBlank#description'
>>> raisins.description
'Golden raisins'
>>> getattr(raisins, '_NonBlank#description')
'Golden raisins'
```

Não foi tão complicado. Decoradores de classe são uma maneira mais simples de fazer algo que antes exigia uma metaclasses: personalizar uma classe no momento em que ela é criada.

Uma desvantagem importante dos decoradores de classe é que eles atuam somente na classe em que são diretamente aplicados. Isso quer dizer que subclasses da classe decorada podem ou não herdar as alterações feitas pelo decorador, dependendo da natureza das alterações. Exploraremos o problema e veremos como resolvê-lo nas seções a seguir.

O que acontece quando: tempo de importação versus tempo de execução

Para uma metaprogramação bem-sucedida, você precisa estar ciente de quando o interpretador Python avalia cada bloco de código. Programadores Python falam de “tempo de importação” (import time) *versus* “tempo de execução” (runtime), mas os termos não são rigorosamente definidos e há uma área cinzenta entre eles. Em tempo de importação, o interpretador faz o parse do código-fonte de um módulo .py em uma passagem do início ao fim e gera o bytecode a ser executado. É nesse momento que os erros de sintaxe podem ocorrer. Se houver um arquivo .pyc atualizado disponível no `__pycache__` local, esses passos serão pulados porque o bytecode já estará pronto para executar.

Embora compilar, definitivamente, seja uma atividade de tempo de importação, outras atividades podem acontecer nesse momento, pois quase todo comando em Python é executável no sentido que pode executar código de usuário e mudar o estado de seu programa. Em particular, o comando `import` não é apenas uma declaração³, mas executa todo o código de nível mais alto do módulo quando ele é importado pela primeira vez no processo – importações subsequentes do mesmo módulo usarão um cache e, nessa ocasião, somente haverá vinculação de nomes aos objetos previamente importados. Esse código de alto nível pode fazer qualquer tarefa, incluindo ações típicas de “tempo de execução”, como conectar-se a um banco de dados.⁴ É por isso que a fronteira entre “tempo de importação” e “tempo de execução” é nebulosa: o comando `import` pode disparar todo tipo de comportamento de “tempo de execução”.

No parágrafo anterior, escrevi que importar “executa todo o código de nível mais alto”, mas “código de nível mais alto” exige algumas explicações. O interpretador executa um comando `def` no nível mais alto de um módulo quando ele é importado, mas o que isso faz? O interpretador compila o corpo da função (se for a primeira vez que o módulo é importado) e vincula o objeto função ao seu nome global, mas obviamente não executa o corpo da função. No caso normal, isso quer dizer que o interpretador define funções de nível mais alto em tempo de importação, mas executa seus corpos somente quando – e se – as funções forem chamadas em tempo de execução.

3 Compare com a instrução `import` em Java, que é apenas uma declaração para que o compilador saiba que determinados pacotes são necessários.

4 Não estou dizendo que iniciar uma conexão com um banco de dados só porque um módulo é importado seja uma boa ideia; estou apenas apontando que isso pode ser feito.

Para classes, a história é diferente: em tempo de importação, o interpretador executa o corpo de todas as classes, até mesmo o corpo das classes aninhadas em outras classes. A execução do corpo de uma classe significa que os atributos e métodos da classe são definidos e o objeto classe propriamente dito é construído. Nesse sentido, o corpo das classes é o “código de nível mais alto”: é executado em tempo de importação.

Tudo isso é muito sutil e abstrato, portanto veja um exercício que ajudará você a ver “o que acontece quando”.

Exercícios dos instantes de avaliação

Considere um script *evaltime.py* que importa um módulo *evalsupport.py*. Os dois módulos têm várias chamadas a `print` para exibir marcadores no formato `<[N]>`, em que `N` é um número. O objetivo desse par de exercícios é determinar quando cada uma dessas chamadas será feita.



Alunos relataram que esses exercícios são úteis para uma melhor apreciação de como Python avalia o código-fonte. Use seu tempo para resolvê-los com lápis e papel antes de ver a seção “Solução para o cenário #1” na página 728.

As listagens estão nos exemplos 21.6 e 21.7. Pegue lápis e papel e – sem executar o código – escreva os marcadores na ordem em que aparecerão na saída, considerando dois cenários:

Cenário #1

O módulo *evaltime.py* é importado interativamente no console de Python:

```
>>> import evaltime
```

Cenário #2

O módulo *evaltime.py* é executado no shell de comandos:

```
$ python3 evaltime.py
```

Exemplo 21.6 – *evaltime.py*: escreva os marcadores `<[N]>` numerados na ordem em que aparecerão na saída

```
from evalsupport import deco_alpha
print('<[1]> evaltime module start')
```

```
class ClassOne():
    print('<[2]> ClassOne body')
    def __init__(self):
```

```

    print('<[3]> ClassOne.__init__')
def __del__(self):
    print('<[4]> ClassOne.__del__')
def method_x(self):
    print('<[5]> ClassOne.method_x')
class ClassTwo(object):
    print('<[6]> ClassTwo body')

@deco_alpha
class ClassThree():
    print('<[7]> ClassThree body')
    def method_y(self):
        print('<[8]> ClassThree.method_y')

class ClassFour(ClassThree):
    print('<[9]> ClassFour body')
    def method_y(self):
        print('<[10]> ClassFour.method_y')

if __name__ == '__main__':
    print('<[11]> ClassOne tests', 30 * '.')
    one = ClassOne()
    one.method_x()
    print('<[12]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[13]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()

print('<[14]> evaltime module end')
```

Exemplo 21.7 – evalsupport.py: módulo importado por evaltime.py

```

print('<[100]> evalsupport module start')
def deco_alpha(cls):
    print('<[200]> deco_alpha')
    def inner_1(self):
        print('<[300]> deco_alpha:inner_1')
    cls.method_y = inner_1
```

```

    return cls

class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2

print('<[700]> evalsupport module end')
```

Solução para o cenário #1

O exemplo 21.8 mostra a saída da importação do módulo *evaltime.py* no console de Python.

Exemplo 21.8 – Cenário #1: importando evaltime no console de Python

```

>>> import evaltime
<[100]> evalsupport module start ❶
<[400]> MetaAleph body ❷
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body ❸
<[6]> ClassTwo body ❹
<[7]> ClassThree body
<[200]> deco_alpha ❺
<[9]> ClassFour body
<[14]> evaltime module end ❻
```

- ❶ Todo código de nível mais alto em evalsupport executa quando o módulo é importado; a função `deco_alpha` é compilada, mas seu corpo não é executado.
- ❷ O corpo da classe `MetaAleph` é executado.
- ❸ O corpo de todas as classes é executado...
- ❹ ... incluindo classes aninhadas.
- ❺ A função decoradora executa depois que o corpo da classe `ClassThree` decorada é avaliado.

❶ Nesse cenário, `evaltime` é importado, portanto o bloco `if __name__ == '__main__':` não executa.

Observações sobre o cenário #1:

- 1. Esse cenário é disparado por um simples comando `import evaltime`.
- 2. O interpretador executa o corpo de todas as classes do módulo importado e de sua dependência, `evalsupport`.
- 3. Faz sentido que o interpretador avalie o corpo de uma classe decorada antes de chamar a função decoradora associada a ela: o decorador precisa receber um objeto-classe para processar, portanto o objeto-classe deve ser criado antes.
- 4. A única função ou o único método definido pelo usuário executado nesse cenário é o decorador `deco_alpha`.

Vamos ver agora o que acontece no cenário #2.

Solução para o cenário #2

O exemplo 21.9 mostra a saída da execução de `python evaltime.py`.

Exemplo 21.9 – Cenário #2: executando `evaltime.py` no shell

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body
<[6]> ClassTwo body
<[7]> ClassThree body
<[200]> deco_alpha
<[9]> ClassFour body ❶
<[11]> ClassOne tests .....
<[3]> ClassOne.__init__ ❷
<[5]> ClassOne.method_x
<[12]> ClassThree tests .....
<[300]> deco_alpha:inner_1 ❸
<[13]> ClassFour tests .....
<[10]> ClassFour.method_y
<[14]> evaltime module end
<[4]> ClassOne.__del__ ❹
```

❶ Mesma saída do exemplo 21.8 até agora.

- ❷ Comportamento-padrão de uma classe.
- ❸ `ClassThree.method_y` foi alterado pelo decorador `deco_alpha`, portanto a chamada a `three.method_y()` executa o corpo da função `inner_1`.
- ❹ A instância de `ClassOne` vinculada à variável global `one` é destruída pelo coletor de lixo somente quando o programa termina.

O ponto principal do cenário #2 é mostrar que os efeitos de um decorador de classe podem não afetar as subclasses. No exemplo 21.6, `ClassFour` é definida como subclasse de `ClassThree`. O decorador `@deco_alpha` é aplicado a `ClassThree`, substituindo seu `method_y`, mas isso não afeta `ClassFour`. É claro que, se `ClassFour.method_y` chamasse `ClassThree.method_y` com `super(...)`, veríamos o efeito do decorador, pois a função `inner_1` seria executada.

Em comparação, a próxima seção mostrará que metaclasses são mais eficazes quando queremos customizar toda uma hierarquia de classes, e não apenas uma classe.

Básico sobre metaclasses

Uma metaclasses é uma fábrica de classes, exceto que, em vez de ser uma função, como `record_factory` do exemplo 21.2, uma metaclasses é escrita como uma classe. A figura 21.1 representa uma metaclasses usando Mills & Gizmos Notation: uma engenhoca produzindo outra engenhoca.

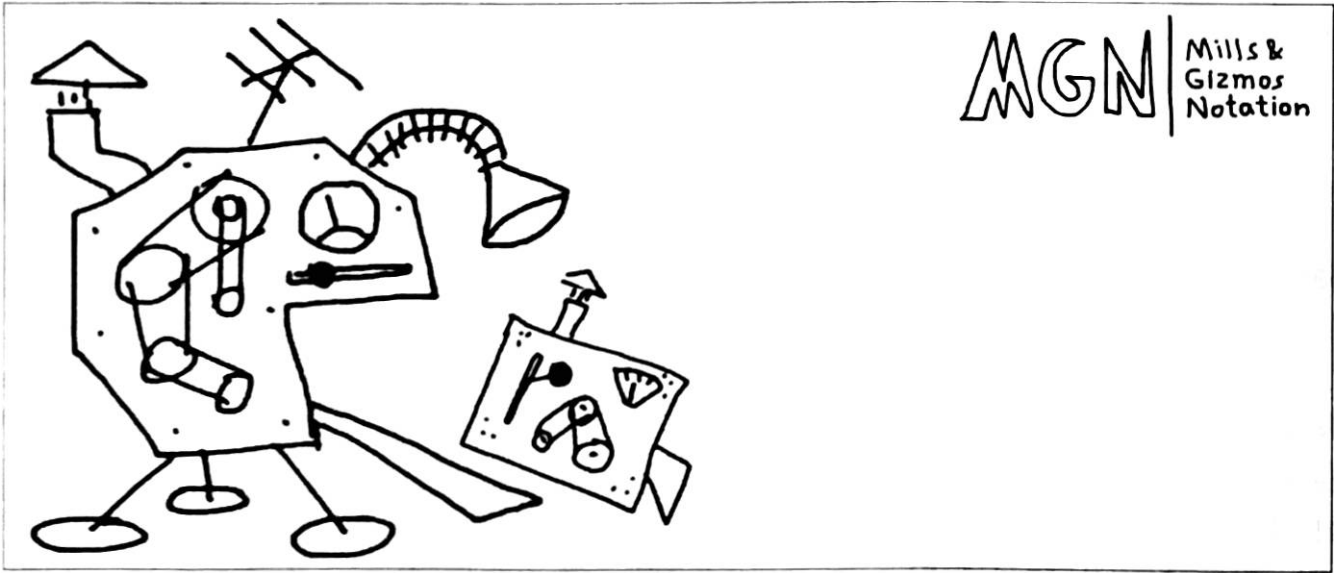


Figura 21.1 – Uma metaclasses é uma classe que cria classes.

Considere o modelo de objetos de Python: classes são objetos, portanto cada classe deve ser uma instância de outra classe. Por padrão, classes em Python são instâncias de `type`. Em outras palavras, `type` é a metaclasses da maioria das classes embutidas e definidas pelo usuário:

```
>>> 'spam'.__class__
<class 'str'>
>>> str.__class__
<class 'type'>
>>> from bulkfood_v6 import LineItem
>>> LineItem.__class__
<class 'type'>
>>> type.__class__
<class 'type'>
```

Para evitar uma regressão infinita, `type` é uma instância de si mesmo, como mostra a última linha.

Observe que não estou dizendo que `str` ou `LineItem` herdam de `type`. O que estou dizendo é que `str` e `LineItem` são instâncias de `type`. Todas elas são subclasses de `object`. A figura 21.2 pode ajudar você a enfrentar essa estranha realidade.

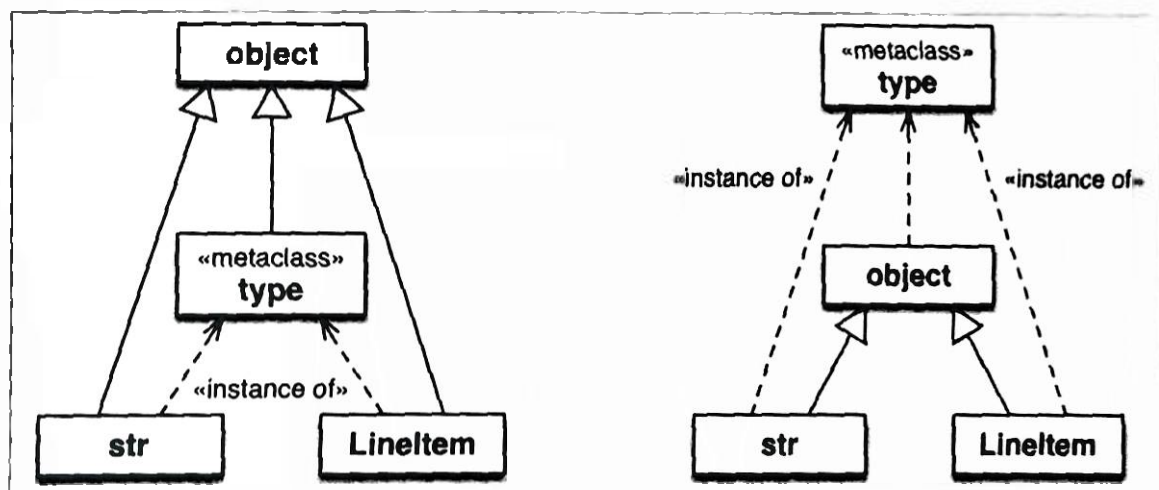


Figura 21.2 – Ambos os diagramas são válidos. O diagrama à esquerda enfatiza que `str`, `type` e `LineItem` são subclasses de `object`. O diagrama da direita deixa claro que `str`, `object` e `LineItem` são instâncias de `type` porque todos são classes.



As classes `object` e `type` têm um relacionamento único: `object` é uma instância de `type` e `type` é uma subclasse de `object`. Esse relacionamento é “mágico”: não pode ser expresso em Python porque cada uma das classes precisaria existir antes de a outra ser definida. O fato de `type` ser uma instância de si mesma também é mágico.

Além de `type`, há outras metaclasses na biblioteca-padrão, como `ABCMeta` e `Enum`. O próximo trecho de código mostra que a metaclasses de `collections.Iterable` é `abc.ABCMeta`. A classe `Iterable` é abstrata, mas `ABCMeta` não é – afinal de contas, `Iterable` é uma instância de `ABCMeta`:

```
>>> import collections
>>> collections.Iterable.__class__
```

```
<class 'abc.ABCMeta'>
>>> import abc
>>> abc.ABCMeta.__class__
<class 'type'>
>>> abc.ABCMeta.__mro__
(<class 'abc.ABCMeta'>, <class 'type'>, <class 'object'>)
```

No fim das contas, a classe de ABCMeta também é uma instância de type. Toda classe é uma instância de type, direta ou indiretamente, mas somente metaclasses são também subclasses de type. Esse é o relacionamento mais importante para entender metaclasses: uma metaclasses, por exemplo, ABCMeta, herda de type o poder de construir classes. A figura 21.3 ilustra esse relacionamento crucial.

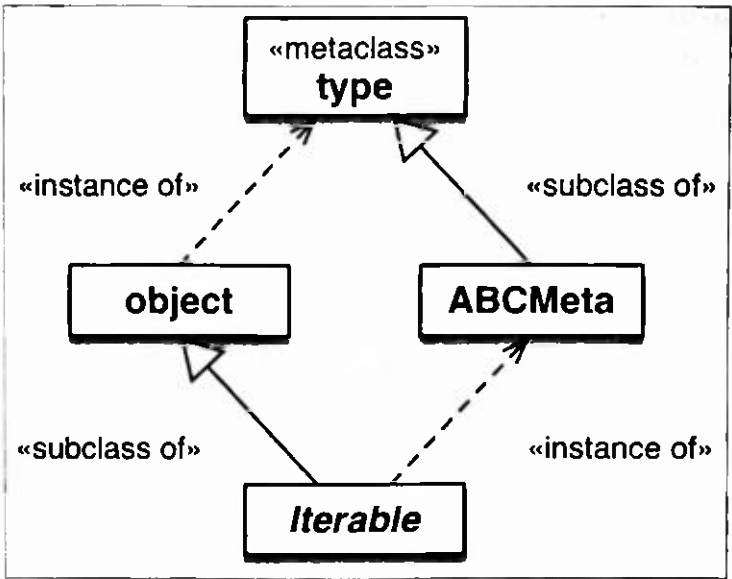


Figura 21.3 – Iterable é uma subclasse de object e uma instância de ABCMeta. Tanto object quanto ABCMeta são instâncias de type, mas o relacionamento essencial aqui é que ABCMeta é também uma subclasse de type porque ABCMeta é uma metaclasses. Nesse diagrama, Iterable é a única classe abstrata.

A lição importante aqui é que todas as classes são instâncias de type, mas as metaclasses são também subclasses de type, portanto atuam como fábricas de classes. Em particular, uma metaclasses pode customizar suas instâncias implementando `__init__` para inicializar cada classe que for construída. O método `__init__` de uma metaclasses pode fazer tudo que um decorador de classe faz, mas seus efeitos são mais profundos, como mostra o próximo exercício.

Exercício do instante de avaliação de metaclasses

Essa é uma variação da seção “Exercícios dos instantes de avaliação” na página 726. O módulo `evalsupport.py` é o mesmo do exemplo 21.7, mas o script principal agora é `evaltime_meta.py`, mostrado no exemplo 21.10.

Exemplo 21.10 – evaltime_meta.py: ClassFive é uma instância da metaclasses MetaAleph

```
from evalsupport import deco_alpha
from evalsupport import MetaAleph

print('<[1]> evaltime_meta module start')

@deco_alpha
class ClassThree():
    print('<[2]> ClassThree body')
    def method_y(self):
        print('<[3]> ClassThree.method_y')

class ClassFour(ClassThree):
    print('<[4]> ClassFour body')
    def method_y(self):
        print('<[5]> ClassFour.method_y')

class ClassFive(metaclass=MetaAleph):
    print('<[6]> ClassFive body')
    def __init__(self):
        print('<[7]> ClassFive.__init__')
    def method_z(self):
        print('<[8]> ClassFive.method_y')

class ClassSix(ClassFive):
    print('<[9]> ClassSix body')
    def method_z(self):
        print('<[10]> ClassSix.method_y')

if __name__ == '__main__':
    print('<[11]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[12]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()
    print('<[13]> ClassFive tests', 30 * '.')
    five = ClassFive()
    five.method_z()
    print('<[14]> ClassSix tests', 30 * '.')
```

```

    six = ClassSix()
    six.method_z()

print('<[15]> evaltime_meta module end')

```

Novamente, pegue lápis e papel e escreva os marcadores <[N]> numerados na ordem em que aparecerão na saída, considerando estes dois cenários:

Cenário #3

O módulo *evaltime_meta.py* é importado interativamente no console de Python.

Cenário #4

O módulo *evaltime_meta.py* é executado no shell de comandos.

As soluções e a análise serão apresentadas a seguir.

Solução para o cenário #3

O exemplo 21.11 mostra a saída da importação de *evaltime_meta.py* no console de Python.

Exemplo 21.11 – Cenário #3: importando *evaltime_meta* no console de Python

```

>>> import evaltime_meta
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__ ❶
<[9]> ClassSix body
<[500]> MetaAleph.__init__ ❷
<[15]> evaltime_meta module end

```

❶ A principal diferença em relação ao cenário #1 é que o método `MetaAleph.__init__` é chamado para inicializar a recém-criada `ClassFive`.

❷ E `MetaAleph.__init__` também inicializa `ClassSix`, que é uma subclasse de `ClassFive`.

O interpretador Python avalia o corpo de `ClassFive`, mas, em vez de chamar `type` para criar o corpo propriamente dito da classe, ele chama `MetaAleph`. Observando a definição de `MetaAleph` no exemplo 21.12, você verá que o método `__init__` recebe quatro argumentos:

`self`

É o objeto classe a ser inicializado (por exemplo, `ClassFive`).

`name, bases, dic`

São os mesmos argumentos passados para `type` para construir uma classe.

Exemplo 21.12 – `evalsupport.py`: definição da metaclasses `MetaAleph` do exemplo 21.7

```
class MetaAleph(type):
    print('<[400]> MetaAleph body')
    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')
        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')
        cls.method_z = inner_2
```



Ao implementar uma metaclasses, substituir `self` por `cls` é uma convenção. Por exemplo, no método `__init__` da metaclasses, usar `cls` como o nome do primeiro argumento deixa claro que a instância em construção é uma classe.

O corpo de `__init__` define uma função `inner_2`, e então ela é vinculada a `cls.method_z`. O nome `cls` na assinatura de `MetaAleph.__init__` refere-se à classe sendo criada (por exemplo, `ClassFive`). Por outro lado, o nome `self` na assinatura de `inner_2` futuramente fará referência a uma instância da classe que estamos criando (por exemplo, uma instância de `ClassFive`).

Solução para o cenário #4

O exemplo 21.13 mostra a saída da execução de `python evaltime.py` na linha de comando.

Exemplo 21.13 – Cenário #4: executando `evaltime_meta.py` no shell

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
```

```
<[500]> MetaAleph.__init__
<[9]> ClassSix body
<[500]> MetaAleph.__init__
<[11]> ClassThree tests .....
<[300]> deco_alpha:inner_1 ❶
<[12]> ClassFour tests .....
<[5]> ClassFour.method_y ❷
<[13]> ClassFive tests .....
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2 ❸
<[14]> ClassSix tests .....
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2 ❹
<[15]> evaltime_meta module end
```

- ❶ Quando o decorador é aplicado a ClassThree, seu method_y é substituído pelo método inner_1...
- ❷ Mas isso não tem efeito na ClassFour não decorada, apesar de ClassFour ser uma subclasse de ClassThree.
- ❸ O método __init__ de MetaAleph substitui ClassFive.method_z pela função inner_2.
- ❹ O mesmo acontece com a subclasse ClassSix de ClassFive: seu method_z é substituído por inner_2.

Observe que ClassSix não faz referência direta a MetaAleph, mas é afetada por ela porque é uma subclasse de ClassFive e, sendo assim, também é uma instância de MetaAleph, portanto é inicializada por MetaAleph.__init__.



Outras customizações de classe podem ser feitas implementando __new__ em uma metaclasses. Mas, com muita frequência, implementar __init__ é suficiente.

Podemos agora colocar toda essa teoria em prática criando uma metaclasses para oferecer uma solução definitiva aos descritores com nomes automáticos para cada atributo de armazenagem.

Uma metaclasses para personalizar descritores

Voltemos aos exemplos de LineItem. Seria bom se o usuário não precisasse estar ciente da existência de decoradores ou de metaclasses e pudesse simplesmente herdar de uma classe fornecida pela nossa biblioteca, como no exemplo 21.14.

Exemplo 21.14 – `bulkfood_v7.py`: herdar de `model.Entity` pode funcionar se uma metaclasses estiver atuando nos bastidores

```
import model_v7 as model

class LineItem(model.Entity): ❶
    description = model.NonBlank()
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ `LineItem` é uma subclasse de `model.Entity`.

O exemplo 21.14 parece bastante inofensivo. Não há nenhuma sintaxe estranha à vista. Contudo ele só funciona porque `model_v7.py` define uma metaclasses e `model.Entity` é uma instância dessa metaclasses. O exemplo 21.15 mostra a implementação da classe `Entity` no módulo `model_v7.py`.

Exemplo 21.15 – `model_v7.py`: a metaclasses `EntityMeta` e uma instância dela, `Entity`

```
class EntityMeta(type):
    """Metaclasses para entidades de negócio com campos validados"""

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict) ❶
        for key, attr in attr_dict.items(): ❷
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{ }#{}'.format(type_name, key)

class Entity(metaclass=EntityMeta): ❸
    """Entidade de negócio com campos validados"""
```

- ❶ Chama `__init__` na superclasse (`type` nesse caso).
- ❷ Mesma lógica do decorador `@entity` do exemplo 21.4.
- ❸ Essa classe existe somente por conveniência: o usuário desse módulo pode simplesmente herdar de `Entity` e não se preocupar com `EntityMeta` – nem sequer tomar conhecimento de sua existência.

O código do exemplo 21.14 passa nos testes do exemplo 21.3. O módulo auxiliar `model_v7.py` é mais difícil de entender que `model_v6.py`, mas o código do nível de usuário é mais simples: basta herdar de `model_v7.Entity` e você terá nomes de armazenagem customizados para cada campo Validated.

A figura 21.4 mostra uma representação simplificada do que acabamos de implementar. Há muita coisa acontecendo, mas a complexidade fica encapsulada no módulo `model_v7`. Do ponto de vista do usuário, `LineItem` é apenas uma subclasse de `Entity`, conforme implementado no exemplo 21.14. Esse é o poder da abstração.

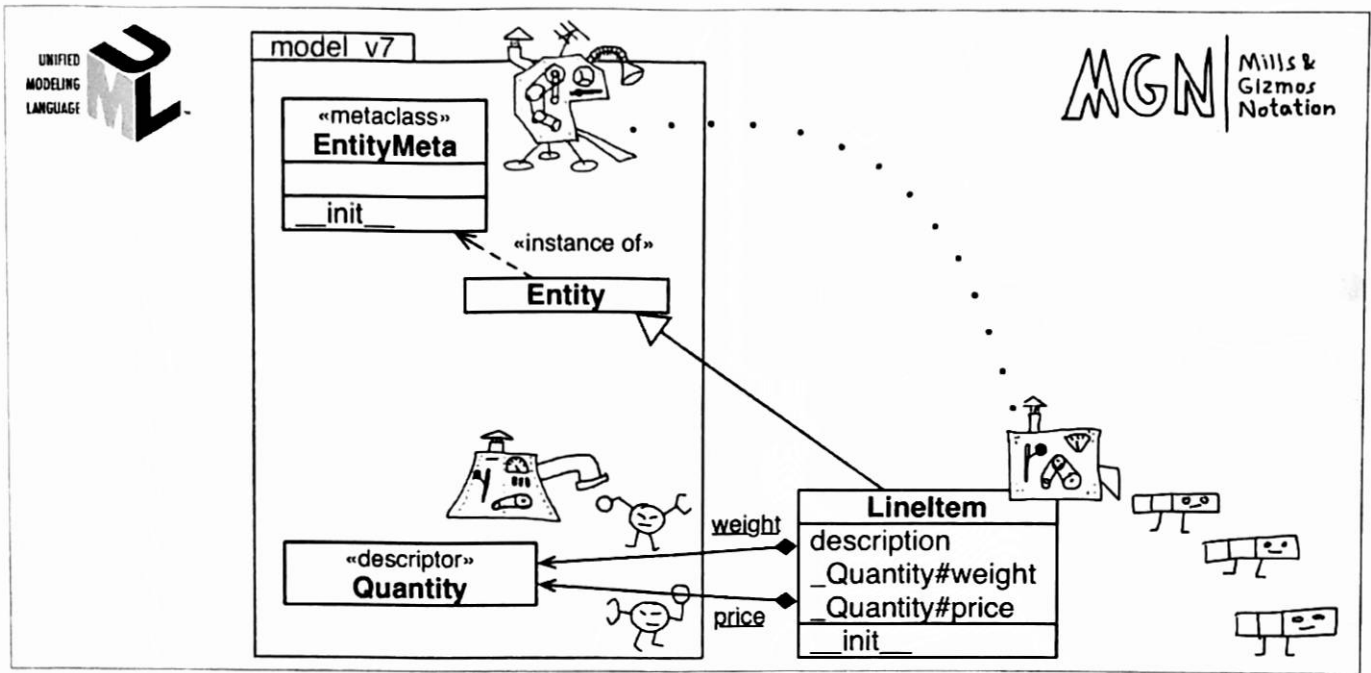


Figura 21.4 – Diagrama de classes UML com anotações em MGN (Mills & Gizmos Notation): a metaengenhoca EntityMeta cria a engenhoca Lineltem. A configuração dos descritores (por exemplo, weight e price) é feita por EntityMeta.___init___.

Exceto pela sintaxe para ligar uma classe à metaclasses,⁵ tudo que foi escrito até agora sobre metaclasses aplica-se a versões de Python desde 2.2, quando os tipos de dados em Python passaram por uma grande mudança. A próxima seção aborda um recurso disponível somente em Python 3.

Método especial `__prepare__` de metaclasses

Em algumas aplicações, é interessante saber a ordem em que os atributos de uma classe são definidos. Por exemplo, uma biblioteca para leitura/escrita de arquivos CSV controlada por classes definidas pelo usuário pode querer mapear a ordem dos campos declarados na classe à ordem das colunas do arquivo CSV.

⁵ Lembre-se de que, de acordo com a seção “Detalhes de sintaxe das ABCs” na página 374, em Python 2.7, o atributo de classe `__metaclass__` é usado e o argumento nomeado `metaclass=` não é aceito na declaração de classe.

Como vimos, tanto o construtor `type` quanto os métodos `__new__` e `__init__` de metaclasses recebem o corpo da classe avaliada como um mapeamento de nomes para atributos. No entanto, por padrão, esse mapeamento é um `dict`, o que significa que a ordem dos atributos como aparecem no corpo da classe estará perdida quando nossa metaclasses ou o decorador de classe puderem olhar para ele.

A solução para esse problema é o método especial `__prepare__`, introduzido em Python 3. Esse método especial é relevante somente em metaclasses e precisa ser um método de classe (ou seja, definido com o decorador `@classmethod`). O método `__prepare__` é chamado pelo interpretador antes do método `__new__` da metaclasses para criar o mapeamento que será preenchido com os atributos do corpo da classe. Além da metaclasses como primeiro argumento, `__prepare__` recebe o nome da classe a ser construída e sua tupla de classes-base e deve devolver um mapeamento, que será recebido como o último argumento de `__new__` e então de `__init__` quando a metaclasses construir uma nova classe.

Na teoria, parece complicado, mas, na prática, sempre que vi `__prepare__` ser usado, era bem simples. Dê uma olhada no exemplo 21.16.

Exemplo 21.16 – `model_v8.py`: a metaclasses `EntityMeta` usa `__prepare__`, e `Entity` agora tem um método de classe `field_names`

```
class EntityMeta(type):
    """Metaclasses para entidades de negócio com campos validados"""

    @classmethod
    def __prepare__(cls, name, bases):
        return collections.OrderedDict() ❶

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)
        cls._field_names = [] ❷
        for key, attr in attr_dict.items(): ❸
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{key}#{type_name}'.format(type_name, key)
                cls._field_names.append(key) ❹

class Entity(metaclass=EntityMeta):
    """Entidade de negócio com campos validados"""

    @classmethod
    def field_names(cls): ❺
        for name in cls._field_names:
            yield name
```

- ❶ Devolve uma instância vazia de `OrderedDict`, no qual os atributos de classe serão armazenados.
- ❷ Cria um atributo `_field_names` na classe em construção.
- ❸ Essa linha permanece inalterada em relação à versão anterior, mas `attr_dict` aqui é o `OrderedDict` obtido pelo interpretador quando chamou `__prepare__` antes de chamar `__init__`. Assim, esse laço `for` percorrerá os atributos na ordem em que foram declarados.
- ❹ Adiciona o nome de cada campo `Validated` encontrado em `_field_names`.
- ❺ O método de classe `field_names` simplesmente produz os nomes dos campos na ordem em que foram declarados.

Com os acréscimos simples feitos no exemplo 21.16, agora podemos iterar pelos campos `Validated` de qualquer subclasse de `Entity` usando o método de classe `field_names`. O exemplo 21.17 mostra esse novo recurso.

Exemplo 21.17 – `bulkfood_v8.py`: `doctest` mostrando o uso de `field_names` – nenhuma alteração é necessária na classe `LineItem`; `field_names` é herdado de `model.Entity`

```
>>> for name in LineItem.field_names():
...     print(name)
...
description
weight
price
```

Com isso, encerramos nossa discussão sobre metaclasses. No mundo real, metaclasses são usadas em frameworks e em bibliotecas que ajudam programadores a executar as seguintes tarefas, entre outras:

- validação de atributos;
- aplicação de decoradores em vários métodos de uma só vez;
- serialização de objetos ou conversão de dados;
- mapeamento objeto-relacional;
- persistência baseada em objetos;
- tradução dinâmica de estruturas de classes de outras linguagens.

Agora apresentaremos uma rápida visão geral dos métodos definidos no modelo de dados de Python para todas as classes.

Classes como objetos

Toda classe tem vários atributos definidos no modelo de dados de Python documentados na seção “4.13. Special Attributes” (Atributos especiais, <http://bit.ly/1cPOodb>) do capítulo “Built-in Types” (Tipos embutidos) da *Library Reference* (Referência à biblioteca). Já vimos três desses atributos várias vezes neste livro: `__mro__`, `__class__` e `__name__`. Outros atributos de classe são:

`cls.__bases__`

A tupla de classes-base da classe.

`cls.__qualname__`

Um novo atributo em Python 3.3 que armazena o nome completo de uma classe ou função, composto do path com ponto a partir do escopo global do módulo até a definição da classe. No exemplo 21.6, `__qualname__` da classe interna `ClassTwo` é a string `'ClassOne.ClassTwo'`, enquanto seu `__name__` é apenas `'ClassTwo'`. A especificação desse atributo está na PEP-3155 – *Qualified name for classes and functions* (Nome qualificado para classes e funções, <http://www.python.org/dev/peps/pep-3155>).

`cls.__subclasses__()`

Esse método devolve uma lista das subclasses imediatas da classe. A implementação usa referências fracas (weak references) para evitar referências circulares entre a superclasse e suas subclasses – que armazenam uma referência forte às superclasses em seu atributo `__bases__`. O método devolve a lista de subclasses existente na memória no momento.

`cls.mro()`

O interpretador chama esse método quando cria uma classe para obter a tupla de superclasses armazenada no atributo `__mro__` da classe. Uma metaclasses pode sobrescrever esse método para customizar a ordem de resolução de métodos da classe em construção.



Nenhum dos atributos mencionados nesta seção é listado pela função `dir(...)`.

Com isso, nosso estudo sobre metaprogramação de classes termina. É um assunto amplo, e mal toquei a superfície. É para isso que temos as seções “Leituras complementares” neste livro.

Resumo do capítulo

Metaprogramação de classes tem a ver com criar ou customizar classes dinamicamente. Classes em Python são objetos de primeira classe, portanto iniciamos o capítulo mostrando como uma classe pode ser criada por uma função chamando a metaclasses embutida `type`.

Na seção seguinte, voltamos à classe `LineItem` com descritores do capítulo 20 para resolver um problema pendente: como gerar nomes para os atributos de armazenagem que refletissem os nomes dos atributos gerenciados (por exemplo, `_Quantity#price` em vez de `_Quantity#1`). A solução foi usar um decorador de classe – essencialmente, uma função que recebe uma classe recém-criada e tem a oportunidade de inspecioná-la, alterá-la e até mesmo substituí-la por uma classe diferente.

Em seguida, passamos para uma discussão sobre quando partes diferentes do código-fonte de um módulo realmente executam. Vimos que há certa sobreposição entre os chamados “tempo de importação” e “tempo de execução”, mas, claramente, a execução de muitos códigos é disparada pelo comando `import`. É fundamental entender “o que é executado quando”, e há algumas regras sutis, de modo que usamos os exercícios de instantes de avaliação para abordar esse tópico.

O próximo assunto foi uma introdução às metaclasses. Vimos que todas as classes são instâncias de `type`, direta ou indiretamente, portanto essa é a “metaclasses-raiz” da linguagem. Uma variação do exercício de instante de avaliação foi concebida para mostrar que uma metaclasses pode personalizar uma hierarquia de classes – em comparação com um decorador de classe, que afeta uma única classe e pode não ter impacto sobre seus descendentes.

A primeira aplicação prática de uma metaclasses foi resolver o problema dos nomes dos atributos de armazenagem em `LineItem`. O código resultante é um pouco mais complicado que a solução com o decorador de classe, mas pode ser encapsulado em um módulo; assim o usuário simplesmente cria subclasses de uma classe aparentemente comum (`model.Entity`), sem ter ciência de que ela é uma instância de uma metaclasses personalizada (`model.EntityMeta`). O resultado final lembra as APIs de ORM em Django e em SQLAlchemy, que usam metaclasses em suas implementações sem exigir que o usuário saiba algo sobre elas.

A segunda metaclasses que implementamos acrescentou um pequeno recurso a `model.EntityMeta`: um método `__prepare__` para fornecer um `OrderedDict` que serve como o mapeamento de nomes para atributos. Isso preserva a ordem em que esses atributos são vinculados no corpo da classe em construção, de modo que métodos da metaclasses, como `__new__` e `__init__`, podem usar essa informação. No exemplo, implementamos

um atributo de classe `_field_names`, que possibilitou ter um `Entity.field_names()`; assim os usuários podem recuperar os descritores `Validated` na mesma ordem em que aparecem no código-fonte.

A última seção apresentou uma rápida visão geral dos atributos e métodos disponíveis em todas as classes Python.

Metaclasses são desafiadoras, empolgantes, e – às vezes – os programadores abusam delas, querendo ser espertos demais. Para encerrar, vamos recordar o último conselho de Alex Martelli em seu texto “Aves aquáticas e ABCs” na página 360:

E não defina ABCs próprias (ou metaclasses) em código de produção... se sentir vontade de fazer isso, aposto que será um caso de síndrome de “todos os problemas se parecem com um prego” para alguém que acabou de ganhar um martelo novinho em folha – você e os futuros mantenedores de seu código serão mais felizes atendo-se a códigos descomplicados e simples, evitando mergulhar nessas profundezas.

— Alex Martelli

Sábias palavras de um homem que não só é um mestre em metaprogramação em Python como também é um engenheiro de software com muitas realizações, que trabalha em alguns dos maiores sistemas de missão crítica em Python no mundo.

Leituras complementares

As referências essenciais para este capítulo na documentação de Python são a seção “3.3.3. Customizing class creation” (Personalizando a criação de classes, <http://bit.ly/1HGwGnI>) do capítulo “Data Model” (Modelo de dados) de *The Python Language Reference* (Guia de referência à linguagem Python), a documentação da classe `type` (<https://docs.python.org/3/library/functions.html#type>) na página “Built-in Functions” (Funções embutidas) e a seção “4.13. Special Attributes” (Atributos especiais, <http://bit.ly/1cPOodb>) do capítulo “Built-in Types” (Tipos embutidos) em *Library Reference* (Referência à biblioteca). Além disso, na *Library Reference*, a documentação do módulo `types` (<http://bit.ly/1HGwF3b>) inclui duas funções que são novas em Python 3.3 e foram projetadas para ajudar em metaprogramação de classes: `types.new_class()` e `types.prepare_class()`.

Decoradores de classe foram formalizados na PEP 3129 – *Class Decorators* (Decoradores de classe, <http://bit.ly/1HGwlvW>), escrita por Collin Winter, com a implementação de referência por Jack Diederich. A palestra “Class Decorators: Radically Simple” (Decoradores de classe: radicalmente simples, vídeo em <http://bit.ly/1HGwJ2Y>) na PyCon 2009, também de Jack Diederich, tem uma introdução rápida a esse recurso.

O livro *Python in a Nutshell*, 2E de Alex Martelli apresenta uma excelente discussão sobre metaclasses, incluindo uma metaclasses `metaMetaBunch` cujo propósito é resolver o mesmo problema de nossa `record_factory` simples do exemplo 21.1, mas é bem mais sofisticada. Martelli não aborda decoradores de classe porque o recurso apareceu depois que seu livro foi escrito. Beazley e Jones oferecem exemplos excelentes de decoradores de classe e de metaclasses em seu livro *Python Cookbook*, 3E (O'Reilly)⁶. Michael Foord escreveu um post intrigante cujo título é “Meta-classes Made Easy: Eliminating self with Metaclasses” (Simplificando metaclasses: eliminando self com metaclasses, <http://bit.ly/1HGwMvx>). O subtítulo diz tudo.

Para metaclasses, as principais referências são a *PEP 3115 – Metaclasses in Python 3000* (Metaclasses em Python 3000, <https://www.python.org/dev/peps/pep-3115/>), em que o método especial `__prepare__` foi introduzido, e *Unifying types and classes in Python 2.2* (Unificando tipos e classes em Python 2.2, <http://bit.ly/1HGwN2D>) de Guido van Rossum. O texto aplica-se a Python 3 também e inclui o que na época foi chamado de semântica de classes “new-style”, incluindo descritores e metaclasses. É uma leitura obrigatória. Uma das referências citadas por Guido é *Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming*, de Ira R. Forman e Scott H. Danforth (Addison-Wesley, 1998), um livro para o qual ele concedeu cinco estrelas na Amazon.com, acrescentando a seguinte avaliação:

Este livro contribuiu para o design de metaclasses em Python 2.2.

Pena que a edição está esgotada; continuo voltando a ele como o melhor tutorial que conheço para o difícil assunto que é a herança múltipla cooperativa, que Python suporta por meio da função `super()`.⁷

Para Python 3.5 (em versão alpha quando escrevi este capítulo), a *PEP 487 – Simpler customization of class creation* (Personalização mais simples de criação de classe, <https://www.python.org/dev/peps/pep-0487/>) propõe um novo método especial `__init_subclass__`, que permitirá a uma classe comum (isto é, que não seja uma metaclasses) personalizar a inicialização de suas subclasses. Como ocorre com decoradores de classe, `__init_subclass__` deixará a metaprogramação de classes mais acessível, além de tornar mais difícil justificar o acionamento de armas nucleares – metaclasses.

Se estiver interessado em metaprogramação, talvez queira que Python tenha o recurso que é a última palavra em metaprogramação: macros sintáticas, como oferecidas por Elixir e as linguagens da família Lisp. Cuidado com o que você deseja. Vou dizer apenas uma palavra: MacroPy (<https://github.com/lihaoyi/macropy>).

6 N.T.: Tradução brasileira publicada pela editora Novatec (<http://novatec.com.br/livros/python-cookbook/>).

7 Página do catálogo da Amazon.com para *Putting Metaclasses to Work* (<http://amzn.to/1HGwKDO>). Você pode comprar um exemplar usado. Comprei esse livro e achei a leitura difícil, mas provavelmente a retomarei no futuro.

Ponto de vista

Iniciarei o último “Ponto de vista” do livro com uma longa citação de Brian Harvey e Matthew Wright, dois professores de ciência da computação da Universidade da Califórnia (Berkeley e Santa Bárbara). Em seu livro *Simply Scheme*, Harvey e Wright escreveram o seguinte:

Há duas escolas de pensamento quando se trata de ensinar ciência da computação. Podemos caricaturar as duas visões assim:

1. A visão conservadora: programas de computador se tornaram muito grandes e complexos para caberem em uma mente humana. Sendo assim, a tarefa do educador em ciência da computação é ensinar as pessoas a disciplinarem seus trabalhos de modo que 500 programadores medíocres possam se reunir e produzir um programa que atenda corretamente à sua especificação.
2. A visão radical: programas de computador se tornaram muito grandes e complexos para caberem em uma mente humana. Sendo assim, a tarefa do educador em ciência da computação é ensinar as pessoas a expandir suas mentes para que os programas possam caber nelas, aprendendo a pensar com um vocabulário composto de ideias mais amplas, mais poderosas e mais flexíveis que as ideias óbvias. Cada unidade de pensamento na programação deve compensar bastante em termos das capacidades do programa.⁸

— Brian Harvey e Matthew Wright
Prefácio do livro Simply Scheme

As descrições exageradas de Harvey e Wright são sobre ensinar ciência da computação, mas também se aplicam ao design de linguagens de programação. A essa altura, você já deve ter adivinhado que sou adepto da visão “radical” e percebido que acredito que Python foi projetado nesse espírito.

A ideia de propriedade é um grande passo à frente se comparada com a abordagem de “métodos de acesso desde o início”, praticamente exigida por Java, com suporte de IDEs que geram getters/setters por meio de um atalho de teclado. A principal vantagem das propriedades é permitir que iniciemos nossos programas simplesmente expondo atributos como públicos – no espírito de *KISS* – sabendo que um atributo público pode se transformar em uma propriedade a qualquer momento, sem muito sofrimento. Mas a ideia de descritor vai muito além, oferecendo um framework para abstrair a lógica repetitiva dos métodos de acesso. Esse framework é tão eficaz que é usado internamente por construções essenciais de Python.

⁸ Brian Harvey e Matthew Wright, *Simply Scheme* (MIT Press, 1999), p. xvii. Texto completo disponível em Berkeley.edu (<https://www.eecs.berkeley.edu/~bh/ss-toc2.html>).

Outra ideia poderosa são funções como objetos de primeira classe, abrindo caminho para funções de ordem superior. A combinação de descritores e funções de ordem superior permite a unificação entre funções e métodos. O `__get__` de uma função produz um objeto método durante a execução, vinculando a instância ao argumento `self`. Isso é elegante.⁹

Por fim, temos a ideia de classes como objetos de primeira classe. É um grande feito de design termos uma linguagem amigável para iniciantes, mas que também oferece abstrações poderosas, como decoradores de classes e metaclasses completas definidas pelo usuário. O melhor de tudo é que os recursos avançados estão integrados de tal modo que Python continua adequado para uma programação casual (esses recursos, na verdade, ajudam, de forma transparente para o usuário). A conveniência e o sucesso de frameworks como Django e SQLAlchemy devem muito às metaclasses, apesar de muitos usuários dessas ferramentas não terem ciência delas. Mas esses usuários sempre podem aprender e criar a próxima grande biblioteca.

Ainda não encontrei uma linguagem que consiga ser fácil para iniciantes, conveniente para profissionais e empolgante para hackers do modo como Python o faz. Obrigado, Guido van Rossum e a todos os demais que a fizeram ser assim.

⁹ *Machine Beauty* de David Gelernter (Basic Books) é um intrigante pequeno livro sobre elegância e estética em obras de engenharia, desde pontes a software.

Python é uma linguagem para adultos com consentimento mútuo.

— Alan Runyan
Cofundador do Plone

A eloquente definição de Alan expressa uma das melhores qualidades de Python: ela sai do caminho e deixa você fazer o que é preciso. Isso também significa que ela não oferece ferramentas para restringir o que outras pessoas podem fazer com seu código e os objetos que ele cria.

É claro que Python não é uma linguagem perfeita. Um dos aspectos mais irritantes para mim é o uso inconsistente de `CamelCase`, `snake_case` e `joinedwords` na biblioteca-padrão. Mas a definição da linguagem e a biblioteca-padrão são apenas parte de um ecossistema. A comunidade de usuários e de colaboradores é a melhor parte do ecossistema de Python.

Veja um exemplo do que há de melhor na comunidade: certa manhã, enquanto escrevia sobre o `asyncio`, senti-me frustrado porque a API tem muitas funções, muitas das quais são corrotinas, e é preciso chamar as corrotinas com `yield from`, mas você não pode fazer isso com funções comuns. Isso estava documentado nas páginas de `asyncio`, mas, às vezes, era preciso ler alguns parágrafos para descobrir se uma função em particular era uma corrotina. Então enviei uma mensagem para `python-tulip` com o título “Proposal: make coroutines stand out in the asyncio docs” (Proposta: fazer as corrotinas se destacarem na documentação de `asyncio`, <https://groups.google.com/forum/#!topic/python-tulip/Y4bhLNbKs74>). Victor Stinner, um core developer de `asyncio`, Andrew Svetlov, autor principal de `aiohttp`, Ben Darnell, desenvolvedor líder do Tornado, e Glyph Lefkowitz, criador do Twisted, juntaram-se à conversa. Darnell sugeriu uma solução, Alexander Shorin explicou como implementá-la no sistema de documentação Sphinx, e Stinner codou as configurações e marcações necessárias. Menos de 12 horas após eu ter levantado a questão, toda a documentação online de `asyncio` havia sido atualizada com as indicações `coroutine` (<https://docs.python.org/3/library/asyncio-eventloop.html#executor>) que podem ser vistas hoje.

Essa história não aconteceu em um clube fechado. Qualquer pessoa pode participar da lista `python-tulip`, e eu havia feito apenas algumas poucas postagens quando escrevi

a proposta. A história mostra uma comunidade realmente aberta a novas ideias e a novos participantes. Guido van Rossum está sempre em python-tulip e pode ser visto frequentemente respondendo até mesmo a perguntas simples.

Outro exemplo de receptividade: a Python Software Foundation (PSF) vem trabalhando para aumentar a diversidade na comunidade Python. Alguns resultados estimulantes já apareceram. O conselho da PSF em 2013–2014 contou com a presença das primeiras mulheres eleitas como diretoras: Jessica McKellar e Lynn Root. Na PyCon América do Norte de 2015 em Montreal – presidida por Diana Clarke –, aproximadamente um terço das palestras foi apresentado por mulheres. Não conheço nenhuma outra grande conferência de TI que tenha ido tão longe na busca da igualdade de gêneros.

Se você é pythonista, mas não está engajado na comunidade, faça isso. Procure o PUG ou GruPy (Python Users Group, ou Grupo de Usuários de Python) de sua região. Se não houver, crie um. Python está em toda parte, portanto você não estará sozinho. Viaje para participar de eventos, se puder. Venha para a conferência PythonBrasil – temos palestrantes internacionais regularmente há vários anos. Conhecer colegas pythonistas pessoalmente é muito melhor que qualquer interação online e, comprovadamente, traz verdadeiros benefícios, além de todo o compartilhamento de conhecimentos. Benefícios como empregos reais e amizades verdadeiras.

Sei que não poderia ter escrito este livro sem a ajuda de muitos amigos que fiz ao longo dos anos na comunidade Python.

Meu pai, Jairo Ramalho, costumava dizer: “Só erra quem trabalha” – um ótimo conselho para não se deixar paralisar pelo medo de cometer erros. Com certeza, cometi minha quota de erros quando escrevi este livro. Os revisores, editores e leitores da Early Release (versão preliminar) identificaram vários deles. Em questão de horas após lançar a Early Release, um leitor já informava erros de digitação na página de errata do livro (<http://www.oreilly.com/catalog/errata.csp?isbn=0636920032519>)¹. Outros leitores contribuíram com mais informações, e amigos entraram em contato direto comigo para oferecer sugestões e correções. Os revisores gramaticais da O'Reilly identificarão outros erros durante o processo de produção, que começará assim que eu conseguir parar de escrever. Assumo total responsabilidade e peço desculpas por quaisquer erros que restarem ou por uma prosa que deixe a desejar.

Estou muito feliz em concluir esta obra, com erros e tudo o mais, e sou muito grato a todos os que me ajudaram no caminho.

Espero ver você em breve em algum evento ao vivo. Por favor, venha dizer olá se me vir por aí!

¹ N.T.: endereço da edição original, em inglês.

Leituras complementares

Encerrarei o livro com referências para o que é “pythônico” – a principal questão que este livro tentou abordar.

Brandon Rhodes é um fantástico professor de Python, e sua palestra “A Python Aesthetic: Beauty and Why I Python” (Uma estética Python: beleza, e por que uso Python, <https://www.youtube.com/watch?v=x-kB2o8sd5c>) é muito bela, a começar pelo uso do caractere Unicode U+00C6 (LATIN CAPITAL LETTER AE) no título. Outro professor incrível, Raymond Hettinger, falou da beleza de Python na PyCon US 2013: “Transforming Code into Beautiful, Idiomatic Python” (Transformando código em Python bonito e idiomático, [https://www.youtube.com/watch?v=\(\)SGv2VnCOgo](https://www.youtube.com/watch?v=()SGv2VnCOgo)).

Vale a pena ler a discussão *Evolution of Style Guides* (Evolução dos guias de estilo, <http://bit.ly/1e8pV4h>) iniciada por Ian Lee na lista Python-ideas. Lee é o mantenedor do pacote pep8 (<https://pypl.python.org/pypl/pep8/>) que verifica códigos-fontes em Python para ver se estão em conformidade com a PEP 8. Para conferir o código deste livro, usei flake8 (<https://pypl.python.org/pypl/flake8/>), que encapsula pep8, pyflakes (<https://pypl.python.org/pypl/pyflakes/>) e o plug-in de complexidade de McCabe (<https://pypl.python.org/pypl/mccabe/>) de Ned Batchelder.

Além da PEP 8, outros guias de estilo influentes são o *Google Python Style Guide* (Guia de estilo de Python do Google, <http://google.github.io/styleguide/pyguide.html>) e o *Pocoo style guide* (Guia de estilo do Pocoo, <http://www.pocoo.org/internal/styleguide/>), da equipe que criou Flake, Sphinx, Jinja 2 e outras ótimas bibliotecas.

The Hitchhiker's Guide to Python! (Guia do mochileiro para Python!, <http://docs.python-guide.org/en/latest/>) é um trabalho coletivo sobre como escrever código pythônico. Seu colaborador mais produtivo é Kenneth Reitz, um herói da comunidade graças ao seu pacote requests, lindamente pythônico. David Goodger apresentou um tutorial na PyCon US 2008 chamado “Code Like a Pythonista: Idiomatic Python” (Escreva como um pythonista: Python idiomático, <http://bit.ly/1e8r8sj>). Quando impressas, as notas do tutorial têm 30 páginas. É claro que o código-fonte das notas em reStructuredText está disponível e pode ser renderizado para HTML e slides S5 (<http://meyerweb.com/eric/tools/s5/>) por docutils. Afinal de contas, Goodger criou tanto reStructuredText quanto docutils – as bases de Sphinx, o excelente sistema de documentação de Python [que, a propósito, também é o sistema de documentação oficial (<http://bit.ly/1e8r4ss>) do MongoDB e de muitos outros projetos].

Martijn Faassen encara de frente a pergunta “O que é pythônico?” (<http://blog.startifact.com/posts/older/what-is-pythonic.html>). Em python-list, há uma discussão com o mesmo título (<http://bit.ly/1e8raAA>). O post de Martijn é de 2005 e a discussão é de 2003, mas o ideal pythônico não mudou muito – nem a linguagem. Uma ótima discussão com

“Pythonic” no título é “Pythonic way to sum n-th list element?” (Maneira pythônica de somar o enésimo elemento de uma lista?, <http://bit.ly/1e8reQP>), da qual extraí muitas citações na seção “Ponto de vista” do capítulo 10 (página 347).

A PEP 3099 – *Things that will Not Change in Python 3000* (Coisas que não mudarão em Python 3000, <https://www.python.org/dev/peps/pep-3099/>) explica por que muitas coisas são como são, mesmo depois da grande mudança em Python 3. Por muito tempo, Python 3 foi apelidada de Python 3000, mas ela chegou alguns séculos antes – para a consternação de algumas pessoas. A PEP 3099 foi escrita por Georg Brandl e compila muitas opiniões expressas pelo BDFL Guido van Rossum. A página *Python Essays* (Artigos sobre Python, <https://www.python.org/doc/essays/>) lista vários textos do próprio Guido.

APÊNDICE A

Scripts auxiliares

A seguir estão as listagens completas de alguns scripts longos demais para serem exibidos no texto principal. Também estão incluídos os scripts usados para gerar algumas tabelas e dados usados neste livro.

Esses scripts também estão disponíveis no repositório de código de *Python fluente* (<https://github.com/fluentpython/example-code>), juntamente com quase todos os demais trechos de código que aparecem no livro.

Capítulo 3: teste de desempenho do operador in

O exemplo A.1 mostra o código que usei para gerar os tempos da tabela 3.6 usando o módulo `timeit`. Em sua maior parte, o script lida com a criação das amostras `haystack` e `needles` e com a formatação da saída.

Enquanto implementava o exemplo A.1, descobri algo que realmente coloca o desempenho de `dict` em perspectiva. Se o script é executado em “modo verboso” (com a opção de linha de comando `-v`), os tempos resultantes são quase o dobro daqueles da tabela 3.5. Mas observe que, nesse script, “modo verboso” implica somente quatro chamadas a `print` na preparação do teste e um `print` adicional para mostrar o número de agulhas (`needles`) encontradas quando cada teste termina. Não há apresentação de saída no laço que faz a busca das agulhas no palheiro (`haystack`), mas essas cinco chamadas a `print` gastam quase tanto tempo quanto procurar mil agulhas.

Exemplo A.1 – `container_perftest.py`: execute-o com o nome de um tipo embutido de coleção como argumento de linha de comando (por exemplo, `container_perftest.py dict`)

```
"""
Teste de desempenho do operador ``in`` em coleções
"""
import sys
```

```

import timeit

SETUP = '''
import array
selected = array.array('d')
with open('selected.arr', 'rb') as fp:
    selected.fromfile(fp, {size})
if {container_type} is dict:
    haystack = dict.fromkeys(selected, 1)
else:
    haystack = {container_type}(selected)
if {verbose}:
    print(type(haystack), end=' ')
    print('haystack: %10d' % len(haystack), end=' ')
needles = array.array('d')
with open('not_selected.arr', 'rb') as fp:
    needles.fromfile(fp, 500)
needles.extend(selected[::{size}//500])
if {verbose}:
    print(' needles: %10d' % len(needles), end=' ')
'''

TEST = '''
found = 0
for n in needles:
    if n in haystack:
        found += 1
if {verbose}:
    print(' found: %10d' % found)
'''

def test(container_type, verbose):
    MAX_EXPONENT = 7
    for n in range(3, MAX_EXPONENT + 1):
        size = 10**n
        setup = SETUP.format(container_type=container_type,
                              size=size, verbose=verbose)
        test = TEST.format(verbose=verbose)
        tt = timeit.repeat(stmt=test, setup=setup, repeat=5, number=1)
        print('|{:{}d}|{:f}'.format(size, MAX_EXPONENT + 1, min(tt)))

if __name__ == '__main__':
    if '-v' in sys.argv:
        sys.argv.remove('-v')
        verbose = True

```

```

else:
    verbose = False
if len(sys.argv) != 2:
    print('Usage: %s <container_type>' % sys.argv[0])
else:
    test(sys.argv[1], verbose)

```

O script *container_perftest_datagen.py* (Exemplo A.2) gera os dados para o script no exemplo A.1.

Exemplo A.2 – *container_perftest_datagen.py*: gera arquivos com arrays de números de ponto flutuante únicos para uso no exemplo A.1

```

"""
Gera dados para teste de desempenho de coleções
"""

import random
import array

MAX_EXPONENT = 7
HAYSTACK_LEN = 10 ** MAX_EXPONENT
NEEDLES_LEN = 10 ** (MAX_EXPONENT - 1)
SAMPLE_LEN = HAYSTACK_LEN + NEEDLES_LEN // 2

needles = array.array('d')

sample = {1/random.random() for i in range(SAMPLE_LEN)}
print('initial sample: %d elements' % len(sample))

# completa a amostra, caso números aleatórios duplicados tenham sido descartados
while len(sample) < SAMPLE_LEN:
    sample.add(1/random.random())

print('complete sample: %d elements' % len(sample))

sample = array.array('d', sample)
random.shuffle(sample)

not_selected = sample[:NEEDLES_LEN // 2]
print('not selected: %d samples' % len(not_selected))
print('  writing not_selected.arr')
with open('not_selected.arr', 'wb') as fp:
    not_selected.tofile(fp)

selected = sample[NEEDLES_LEN // 2:]
print('selected: %d samples' % len(selected))
print('  writing selected.arr')

```

```
with open('selected.arr', 'wb') as fp:
    selected.tofile(fp)
```

Capítulo 3: comparar padrões de bits de hashes

O exemplo A.3 é um script simples para mostrar como são diferentes os padrões de bits para as hashes de números de ponto flutuante semelhantes (por exemplo, 1,0001, 1,0002 etc.). Sua saída está no exemplo 3.16.

Exemplo A.3 – `hashdiff.py`: exibe a diferença entre padrões de bits de valores de hash

```
import sys

MAX_BITS = len(format(sys.maxsize, 'b'))
print('%s-bit Python build' % (MAX_BITS + 1))

def hash_diff(o1, o2):
    h1 = '{:>0{}}b'.format(hash(o1), MAX_BITS)
    h2 = '{:>0{}}b'.format(hash(o2), MAX_BITS)
    diff = ''.join('!' if b1 != b2 else ' ' for b1, b2 in zip(h1, h2))
    count = '!= {}'.format(diff.count('!'))
    width = max(len(repr(o1)), len(repr(o2)), 8)
    sep = '-' * (width * 2 + MAX_BITS)
    return '{!r:{width}} {} \n{: {width}} {} {} \n{!r:{width}} {} \n{}'.format(
        o1, h1, ' ' * width, diff, count, o2, h2, sep, width=width)

if __name__ == '__main__':
    print(hash_diff(1, 1.0))
    print(hash_diff(1.0, 1.0001))
    print(hash_diff(1.0001, 1.0002))
    print(hash_diff(1.0002, 1.0003))
```

Capítulo 9: uso de RAM com e sem `__slots__`

O script *memtest.py* foi usado para uma demonstração na seção “Economizando espaço com o atributo de classe `__slots__`” na página 307: exemplo 9.12.

O script *memtest.py* recebe um nome de módulo na linha de comando e o carrega. Supondo que o módulo defina uma classe chamada *Vector*, *memtest.py* cria uma lista com dez milhões de instâncias, informando o uso de memória antes e depois da criação da lista.

Exemplo A.4 – `memtest.py`: cria muitas instâncias de `Vector`, informando o uso de memória

```
import importlib
import sys
import resource

NUM_VECTORS = 10**7

if len(sys.argv) == 2:
    module_name = sys.argv[1].replace('.py', '')
    module = importlib.import_module(module_name)
else:
    print('Usage: {} <vector-module-to-test>'.format())
    sys.exit(1)

fmt = 'Selected Vector2d type: {.__name__}.{.__name__}'
print(fmt.format(module, module.Vector2d))

mem_init = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
print('Creating {:,} Vector2d instances'.format(NUM_VECTORS))

vectors = [module.Vector2d(3.0, 4.0) for i in range(NUM_VECTORS)]

mem_final = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
print('Initial RAM usage: {:14,}'.format(mem_init))
print('  Final RAM usage: {:14,}'.format(mem_final))
```

Capítulo 14: script `isis2json.py` para conversão de banco de dados

O exemplo A.5 mostra o script `isis2json.py` discutido na seção “Estudo de caso: geradores e um utilitário para conversão de banco de dados” na página 489 (Capítulo 14). Esse script usa funções geradoras para converter bancos de dados CDS/ISIS em JSON em modo preguiçoso (*lazy*), a fim de carregar os dados em CouchDB ou MongoDB.

Observe que esse é um script em Python 2 projetado para executar em CPython ou Jython, versões 2.5 a 2.7, mas não em Python 3. Em CPython, o script lê apenas arquivos `.iso`; com Jython, ele pode ler arquivos `.mst` também usando a biblioteca `Bruma`, disponível no repositório `fluentpython/isis2json` (<https://github.com/fluentpython/isis2json>) no GitHub. Veja a documentação para uso do script nesse repositório.

Exemplo A.5 – `isis2json.py`: dependências e documentação disponíveis no repositório do GitHub em `fluentpython/isis2json`

```
# este script funciona com Python ou Jython (versões >=2.5 e <3)

import sys
import argparse
```

```

from uuid import uuid4
import os

try:
    import json
except ImportError:
    if os.name == 'java': # executando Jython
        from com.xhaus.jyson import JysonCodec as json
    else:
        import simplejson as json

SKIP_INACTIVE = True
DEFAULT_QTY = 2**31
ISIS_MFN_KEY = 'mfn'
ISIS_ACTIVE_KEY = 'active'
SUBFIELD_DELIMITER = '^'
INPUT_ENCODING = 'cp1252'

def iter_iso_records(iso_file_name, isis_json_type): ❶
    from iso2709 import IsoFile
    from subfield import expand

    iso = IsoFile(iso_file_name)
    for record in iso:
        fields = {}
        for field in record.directory:
            field_key = str(int(field.tag)) # remove zeros na frente
            field_occurrences = fields.setdefault(field_key, [])
            content = field.value.decode(INPUT_ENCODING, 'replace')
            if isis_json_type == 1:
                field_occurrences.append(content)
            elif isis_json_type == 2:
                field_occurrences.append(expand(content))
            elif isis_json_type == 3:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError('ISIS-JSON type %s conversion '
                                          'not yet implemented for .iso input' % isis_json_type)

        yield fields
    iso.close()

def iter_mst_records(master_file_name, isis_json_type): ❷
    try:
        from bruma.master import MasterFactory, Record

```



```

except ImportError:
    print('IMPORT ERROR: Jython 2.5 and Bruma.jar '
          'are required to read .mst files')
    raise SystemExit
mst = MasterFactory.getInstance(master_file_name).open()
for record in mst:
    fields = {}
    if SKIP_INACTIVE:
        if record.getStatus() != Record.Status.ACTIVE:
            continue
    else: # salva status somente se houver registros não ativos
        fields[ISIS_ACTIVE_KEY] = (record.getStatus() ==
                                   Record.Status.ACTIVE)
    fields[ISIS_MFN_KEY] = record.getMfn()
    for field in record.getFields():
        field_key = str(field.getId())
        field_occurrences = fields.setdefault(field_key, [])
        if isis_json_type == 3:
            content = {}
            for subfield in field.getSubfields():
                subfield_key = subfield.getId()
                if subfield_key == '*':
                    content['_'] = subfield.getContent()
                else:
                    subfield_occurrences = content.setdefault(subfield_key, [])
                    subfield_occurrences.append(subfield.getContent())
            field_occurrences.append(content)
        elif isis_json_type == 1:
            content = []
            for subfield in field.getSubfields():
                subfield_key = subfield.getId()
                if subfield_key == '*':
                    content.insert(0, subfield.getContent())
                else:
                    content.append(SUBFIELD_DELIMITER + subfield_key +
                                  subfield.getContent())
            field_occurrences.append(''.join(content))
        else:
            raise NotImplementedError('ISIS-JSON type %s conversion '
                                      'not yet implemented for .mst input' % isis_json_type)
    yield fields
mst.close()

```

```

def write_json(input_gen, file_name, output, qty, skip, id_tag, ❸
               gen_uuid, mongo, mfn, isis_json_type, prefix,
               constant):
    start = skip
    end = start + qty
    if id_tag:
        id_tag = str(id_tag)
        ids = set()
    else:
        id_tag = ''
    for i, record in enumerate(input_gen):
        if i >= end:
            break
        if not mongo:
            if i == 0:
                output.write('[')
            elif i > start:
                output.write(',')
        if start <= i < end:
            if id_tag:
                occurrences = record.get(id_tag, None)
                if occurrences is None:
                    msg = 'id tag %s not found in record %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                    raise KeyError(msg % (id_tag, i))
                if len(occurrences) > 1:
                    msg = 'multiple id tags %s found in record %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                    raise TypeError(msg % (id_tag, i))
            else: # ok, temos um e somente um campo id
                if isis_json_type == 1:
                    id = occurrences[0]
                elif isis_json_type == 2:
                    id = occurrences[0][0][1]
                elif isis_json_type == 3:
                    id = occurrences[0]['_']
                if id in ids:
                    msg = 'duplicate id %s in tag %s, record %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])

```

```

        raise TypeError(msg % (id, id_tag, i))
        record['_id'] = id
        ids.add(id)
    elif gen_uuid:
        record['_id'] = unicode(uuid4())
    elif mfn:
        record['_id'] = record[ISIS_MFN_KEY]
    if prefix:
        # itera por uma sequência fixa de tags
        for tag in tuple(record):
            if str(tag).isdigit():
                record[prefix+tag] = record[tag]
                del record[tag] # é por isso que iteramos por uma tupla
                                # com as tags, e não diretamente no dict record
    if constant:
        constant_key, constant_value = constant.split(':')
        record[constant_key] = constant_value
    output.write(json.dumps(record).encode('utf-8'))
    output.write('\n')

if not mongo:
    output.write(']\n')

def main(): ❹
    # cria o parser
    parser = argparse.ArgumentParser(
        description='Convert an ISIS .mst or .iso file to a JSON array')

    # adiciona os argumentos
    parser.add_argument(
        'file_name', metavar='INPUT.(mst|iso)',
        help='.mst or .iso file to read')
    parser.add_argument(
        '-o', '--out', type=argparse.FileType('w'), default=sys.stdout,
        metavar='OUTPUT.json',
        help='the file where the JSON output should be written'
        ' (default: write to stdout)')
    parser.add_argument(
        '-c', '--couch', action='store_true',
        help='output array within a "docs" item in a JSON document'
        ' for bulk insert to CouchDB via POST to db/_bulk_docs')
    parser.add_argument(
        '-m', '--mongo', action='store_true',

```

```

    help='output individual records as separate JSON dictionaries, one'
        ' per line for bulk insert to MongoDB via mongoimport utility')
parser.add_argument(
    '-t', '--type', type=int, metavar='ISIS_JSON_TYPE', default=1,
    help='ISIS-JSON type, sets field structure: 1=string, 2=alist,'
        ' 3=dict (default=1)')
parser.add_argument(
    '-q', '--qty', type=int, default=DEFAULT_QTY,
    help='maximum quantity of records to read (default=ALL)')
parser.add_argument(
    '-s', '--skip', type=int, default=0,
    help='records to skip from start of .mst (default=0)')
parser.add_argument(
    '-i', '--id', type=int, metavar='TAG_NUMBER', default=0,
    help='generate an "_id" from the given unique TAG field number'
        ' for each record')
parser.add_argument(
    '-u', '--uuid', action='store_true',
    help='generate an "_id" with a random UUID for each record')
parser.add_argument(
    '-p', '--prefix', type=str, metavar='PREFIX', default='',
    help='concatenate prefix to every numeric field tag'
        ' (ex. 99 becomes "v99")')
parser.add_argument(
    '-n', '--mfn', action='store_true',
    help='generate an "_id" from the MFN of each record'
        ' (available only for .mst input)')
parser.add_argument(
    '-k', '--constant', type=str, metavar='TAG:VALUE', default='',
    help='Include a constant tag:value in every record (ex. -k type:AS)')
...

# TODO: para exportar grandes cantidades de registros ao CouchDB
parser.add_argument(
    '-r', '--repeat', type=int, default=1,
    help='repeat operation, saving multiple JSON files'
        ' (default=1, use -r 0 to repeat until end of input)')
...

# faz parse da linha de comando
args = parser.parse_args()
if args.file_name.lower().endswith('.mst'):
    input_gen_func = iter_mst_records ❸

```

```

else:
    if args.mfn:
        print('UNSUPPORTED: -n/--mfn option only available for .mst input.')
        raise SystemExit
    input_gen_func = iter_iso_records ❶
input_gen = input_gen_func(args.file_name, args.type) ❷
if args.couch:
    args.out.write('{ "docs" : ')
write_json(input_gen, args.file_name, args.out, args.qty, ❸
           args.skip, args.id, args.uuid, args.mongo, args.mfn,
           args.type, args.prefix, args.constant)
if args.couch:
    args.out.write('}\n')
args.out.close()

if __name__ == '__main__':
    main()

```

- ❶ A função geradora `iter_iso_records` lê um arquivo `.iso` e produz registros.
- ❷ A função geradora `iter_mst_records` lê um arquivo `.mst` e produz registros.
- ❸ `write_json` itera pelo gerador `input_gen` e gera o arquivo `.json`.
- ❹ A função principal lê os argumentos da linha de comando e então...
- ❺ ... seleciona `iter_iso_records` ou...
- ❻ ... `iter_mst_record`, de acordo com a extensão do arquivo de entrada.
- ❼ Um objeto gerador é criado a partir da função geradora selecionada.
- ❽ `write_json` é chamado com o gerador como primeiro argumento.

Capítulo 16: simulação de eventos discretos para a frota de táxis

O exemplo A.6 mostra a listagem completa de *taxi_sim.py*, discutido na seção “A simulação da frota de táxis” na página 544.

Exemplo A.6 – *taxi_sim.py*: o simulador da frota de táxis

```
"""
```

```
Simulador de táxis
```

```
=====
```

```
Dirigindo um táxi a partir do console::
```

```
>>> from taxi_sim import taxi_process
>>> taxi = taxi_process(ident=13, trips=2, start_time=0)
>>> next(taxi)
Event(time=0, proc=13, action='leave garage')
>>> taxi.send(_.time + 7)
Event(time=7, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 23)
Event(time=30, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 5)
Event(time=35, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 48)
Event(time=83, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 1)
Event(time=84, proc=13, action='going home')
>>> taxi.send(_.time + 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Exemplo de execução com dois carros, semente (seed) de aleatoriedade 10. Esse é um doctest válido::

```
>>> main(num_taxis=2, seed=10)
taxi: 0 Event(time=0, proc=0, action='leave garage')
taxi: 0 Event(time=5, proc=0, action='pick up passenger')
taxi: 1 Event(time=5, proc=1, action='leave garage')
taxi: 1 Event(time=10, proc=1, action='pick up passenger')
taxi: 1 Event(time=15, proc=1, action='drop off passenger')
taxi: 0 Event(time=17, proc=0, action='drop off passenger')
taxi: 1 Event(time=24, proc=1, action='pick up passenger')
taxi: 0 Event(time=26, proc=0, action='pick up passenger')
taxi: 0 Event(time=30, proc=0, action='drop off passenger')
taxi: 0 Event(time=34, proc=0, action='going home')
taxi: 1 Event(time=46, proc=1, action='drop off passenger')
taxi: 1 Event(time=48, proc=1, action='pick up passenger')
taxi: 1 Event(time=110, proc=1, action='drop off passenger')
taxi: 1 Event(time=139, proc=1, action='pick up passenger')
taxi: 1 Event(time=140, proc=1, action='drop off passenger')
taxi: 1 Event(time=150, proc=1, action='going home')
*** end of events ***
```

Veja um exemplo mais longo de execução no final deste módulo.

```

"""

import random
import collections
import queue
import argparse
import time

DEFAULT_NUMBER_OF_TAXIS = 3
DEFAULT_END_TIME = 180
SEARCH_DURATION = 5
TRIP_DURATION = 20
DEPARTURE_INTERVAL = 5

Event = collections.namedtuple('Event', 'time proc action')

# BEGIN TAXI_PROCESS
def taxi_process(ident, trips, start_time=0):
    """Cede o controle ao simulador gerando um evento a cada mudança de estado"""
    time = yield Event(start_time, ident, 'leave garage')
    for i in range(trips):
        time = yield Event(time, ident, 'pick up passenger')
        time = yield Event(time, ident, 'drop off passenger')

    yield Event(time, ident, 'going home')
    # fim do processo associado ao táxi
# END TAXI_PROCESS

# BEGIN TAXI_SIMULATOR
class Simulator:
    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)

    def run(self, end_time):
        """Agenda e exibe eventos até o tempo acabar"""
        # agenda o primeiro evento para cada táxi
        for _, proc in sorted(self.procs.items()):
            first_event = next(proc)
            self.events.put(first_event)

        # laço principal da simulação
        sim_time = 0

```

```

while sim_time < end_time:
    if self.events.empty():
        print('*** end of events ***')
        break

    current_event = self.events.get()
    sim_time, proc_id, previous_action = current_event
    print('taxi:', proc_id, proc_id * ' ', current_event)
    active_proc = self.procs[proc_id]
    next_time = sim_time + compute_duration(previous_action)
    try:
        next_event = active_proc.send(next_time)
    except StopIteration:
        del self.procs[proc_id]
    else:
        self.events.put(next_event)
else:
    msg = '*** end of simulation time: {} events pending ***'
    print(msg.format(self.events.qsize()))
# END TAXI_SIMULATOR

def compute_duration(previous_action):
    """Calcula a duração da ação usando distribuição exponencial"""
    if previous_action in ['leave garage', 'drop off passenger']:
        # novo estado é circulando
        interval = SEARCH_DURATION
    elif previous_action == 'pick up passenger':
        # novo estado é fazendo corrida
        interval = TRIP_DURATION
    elif previous_action == 'going home':
        interval = 1
    else:
        raise ValueError('Unknown previous_action: %s' % previous_action)
    return int(random.expovariate(1/interval)) + 1

def main(end_time=DEFAULT_END_TIME, num_taxis=DEFAULT_NUMBER_OF_TAXIS,
        seed=None):
    """Inicializa o gerador aleatório, cria procs e executa a simulação"""
    if seed is not None:
        random.seed(seed) # obtém resultados reproduzíveis
    taxis = {i: taxi_process(i, (i+1)*2, i*DEPARTURE_INTERVAL)
             for i in range(num_taxis)}
    sim = Simulator(taxis)

```


Apêndice A ■ Scripts auxiliares

```

sim.run(end_time)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Taxi fleet simulator.')
    parser.add_argument('-e', '--end-time', type=int,
        default=DEFAULT_END_TIME,
        help='simulation end time; default = %s'
        % DEFAULT_END_TIME)
    parser.add_argument('-t', '--taxis', type=int,
        default=DEFAULT_NUMBER_OF_TAXIS,
        help='number of taxis running; default = %s'
        % DEFAULT_NUMBER_OF_TAXIS)
    parser.add_argument('-s', '--seed', type=int, default=None,
        help='random generator seed (for testing)')

    args = parser.parse_args()
    main(args.end_time, args.taxis, args.seed)

```

Exemplo de execução a partir da linha de comando, seed=3, tempo máximo=120::

```

# BEGIN TAXI_SAMPLE_RUN
$ python3 taxi_sim.py -s 3 -e 120
taxi: 0 Event(time=0, proc=0, action='leave garage')
taxi: 0 Event(time=2, proc=0, action='pick up passenger')
taxi: 1 Event(time=5, proc=1, action='leave garage')
taxi: 1 Event(time=8, proc=1, action='pick up passenger')
taxi: 2 Event(time=10, proc=2, action='leave garage')
taxi: 2 Event(time=15, proc=2, action='pick up passenger')
taxi: 2 Event(time=17, proc=2, action='drop off passenger')
taxi: 0 Event(time=18, proc=0, action='drop off passenger')
taxi: 2 Event(time=18, proc=2, action='pick up passenger')
taxi: 2 Event(time=25, proc=2, action='drop off passenger')
taxi: 1 Event(time=27, proc=1, action='drop off passenger')
taxi: 2 Event(time=27, proc=2, action='pick up passenger')
taxi: 0 Event(time=28, proc=0, action='pick up passenger')
taxi: 2 Event(time=40, proc=2, action='drop off passenger')
taxi: 2 Event(time=44, proc=2, action='pick up passenger')
taxi: 1 Event(time=55, proc=1, action='pick up passenger')
taxi: 1 Event(time=59, proc=1, action='drop off passenger')
taxi: 0 Event(time=65, proc=0, action='drop off passenger')
taxi: 1 Event(time=65, proc=1, action='pick up passenger')

```

```

taxi: 2      Event(time=65, proc=2, action='drop off passenger')
taxi: 2      Event(time=72, proc=2, action='pick up passenger')
taxi: 0      Event(time=76, proc=0, action='going home')
taxi: 1      Event(time=80, proc=1, action='drop off passenger')
taxi: 1      Event(time=88, proc=1, action='pick up passenger')
taxi: 2      Event(time=95, proc=2, action='drop off passenger')
taxi: 2      Event(time=97, proc=2, action='pick up passenger')
taxi: 2      Event(time=98, proc=2, action='drop off passenger')
taxi: 1      Event(time=106, proc=1, action='drop off passenger')
taxi: 2      Event(time=109, proc=2, action='going home')
taxi: 1      Event(time=110, proc=1, action='going home')
*** end of events ***
# END TAXI_SAMPLE_RUN
"""

```

Capítulo 17: Exemplos com criptografia

Esses scripts foram usados para mostrar `futures.ProcessPoolExecutor`, utilizado para executar tarefas com uso intensivo de CPU.

O exemplo A.7 criptografa e descriptografa arrays de bytes aleatórios com o algoritmo RC4. Ele depende do módulo *arcfour.py* (Exemplo A.8) para executar.

Exemplo A.7 – *arcfour_futures.py*: exemplo de `futures.ProcessPoolExecutor`

```

import sys
import time
from concurrent import futures
from random import randrange
from arcfour import arcfour

JOBS = 12
SIZE = 2**18

KEY = b"'Twas brillig, and the slithy toves\nDid gyre"
STATUS = '{} workers, elapsed time: {:.2f}s'

def arcfour_test(size, key):
    in_text = bytearray(randrange(256) for i in range(size))
    cypher_text = arcfour(key, in_text)
    out_text = arcfour(key, cypher_text)
    assert in_text == out_text, 'Failed arcfour_test'
    return size

```

```
def main(workers=None):
    if workers:
        workers = int(workers)
    t0 = time.time()

    with futures.ProcessPoolExecutor(workers) as executor:
        actual_workers = executor._max_workers
        to_do = []
        for i in range(JOBS, 0, -1):
            size = SIZE + int(SIZE / JOBS * (i - JOBS/2))
            job = executor.submit(arcfour_test, size, KEY)
            to_do.append(job)

            for future in futures.as_completed(to_do):
                res = future.result()
                print('{:.1f} KB'.format(res/2**10))

        print(STATUS.format(actual_workers, time.time() - t0))

if __name__ == '__main__':
    if len(sys.argv) == 2:
        workers = int(sys.argv[1])
    else:
        workers = None
    main(workers)
```

O exemplo A.8 implementa o algoritmo de criptografia RC4 em Python puro.

Exemplo A.8 – `arcfour.py`: algoritmo compatível com RC4

```
"""algoritmo compatível com RC4"""

def arcfour(key, in_bytes, loops=20):
    kbox = bytearray(256) # cria a caixa de chaves
    for i, car in enumerate(key): # copia chave e vetor
        kbox[i] = car
    j = len(key)
    for i in range(j, 256): # repete até encher
        kbox[i] = kbox[i-j]

    # [1] inicializa sbbox
    sbbox = bytearray(range(256))

    # repete laço para embaralhar sbbox, conforme recomendado em CipherSaber-2
    # http://ciphersaber.gurus.com/faq.html#cs2
    j = 0
    for k in range(loops):
```

```

    for i in range(256):
        j = (j + sbbox[i] + kbox[i]) % 256
        sbbox[i], sbbox[j] = sbbox[j], sbbox[i]

# laço principal
i = 0
j = 0
out_bytes = bytearray()

for car in in_bytes:
    i = (i + 1) % 256
    # [2] embaralha sbbox
    j = (j + sbbox[i]) % 256
    sbbox[i], sbbox[j] = sbbox[j], sbbox[i]
    # [3] calcula t
    t = (sbbox[i] + sbbox[j]) % 256
    k = sbbox[t]
    car = car ^ k
    out_bytes.append(car)

return out_bytes

def test():
    from time import time
    clear = bytearray(b'1234567890' * 100000)
    t0 = time()
    cipher = arcfour(b'key', clear)
    print('elapsed time: %.2fs' % (time() - t0))
    result = arcfour(b'key', cipher)
    assert result == clear, '%r != %r' % (result, clear)
    print('elapsed time: %.2fs' % (time() - t0))
    print('OK')

if __name__ == '__main__':
    test()

```

O exemplo A.9 aplica o algoritmo de hash SHA-256 a arrays de bytes aleatórios. Usa hashlib da biblioteca-padrão que, por sua vez, utiliza a biblioteca OpenSSL escrita em C.

Exemplo A.9 – sha_futures.py: exemplo de futures.ProcessPoolExecutor

```

import sys
import time
import hashlib
from concurrent import futures

```

```

from random import randrange

JOBS = 12
SIZE = 2**20
STATUS = '{} workers, elapsed time: {:.2f}s'

def sha(size):
    data = bytearray(randrange(256) for i in range(size))
    algo = hashlib.new('sha256')
    algo.update(data)
    return algo.hexdigest()

def main(workers=None):
    if workers:
        workers = int(workers)
    t0 = time.time()

    with futures.ProcessPoolExecutor(workers) as executor:
        actual_workers = executor._max_workers
        to_do = (executor.submit(sha, SIZE) for i in range(JOBS))
        for future in futures.as_completed(to_do):
            res = future.result()
            print(res)

    print(STATUS.format(actual_workers, time.time() - t0))

if __name__ == '__main__':
    if len(sys.argv) == 2:
        workers = int(sys.argv[1])
    else:
        workers = None
    main(workers)

```

Capítulo 17: exemplos de cliente HTTP para flags2

Todos os exemplos da série flags2 da seção “Downloads com exibição de progresso e tratamento de erros” na página 576 usam funções do módulo *flags2_common.py* (Exemplo A.10).

Exemplo A.10 – flags2_common.py

```

"""Utilitários para o segundo conjunto de exemplos com as bandeiras.
"""

import os

```

```

import time
import sys
import string
import argparse
from collections import namedtuple
from enum import Enum

Result = namedtuple('Result', 'status data')

HTTPStatus = Enum('Status', 'ok not_found error')

POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split()

DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1

SERVERS = {
    'REMOTE': 'http://flupy.org/data/flags',
    'LOCAL': 'http://localhost:8001/flags',
    'DELAY': 'http://localhost:8002/flags',
    'ERROR': 'http://localhost:8003/flags',
}

DEFAULT_SERVER = 'LOCAL'

DEST_DIR = 'downloads/'
COUNTRY_CODES_FILE = 'country_codes.txt'

def save_flag(img, filename):
    path = os.path.join(DEST_DIR, filename)
    with open(path, 'wb') as fp:
        fp.write(img)

def initial_report(cc_list, actual_req, server_label):
    if len(cc_list) <= 10:
        cc_msg = ', '.join(cc_list)
    else:
        cc_msg = 'from {} to {}'.format(cc_list[0], cc_list[-1])
    print('{} site: {}'.format(server_label, SERVERS[server_label]))
    msg = 'Searching for {} flag{}: {}'.format(
        len(cc_list), plural, cc_msg)
    plural = 's' if len(cc_list) != 1 else ''
    print(msg.format(len(cc_list), plural, cc_msg))
    plural = 's' if actual_req != 1 else ''
    msg = '{} concurrent connection{} will be used.'
    print(msg.format(actual_req, plural))

```

```

def final_report(cc_list, counter, start_time):
    elapsed = time.time() - start_time
    print('-' * 20)
    msg = '{} flag{} downloaded.'
    plural = 's' if counter[HTTPStatus.ok] != 1 else ''
    print(msg.format(counter[HTTPStatus.ok], plural))
    if counter[HTTPStatus.not_found]:
        print(counter[HTTPStatus.not_found], 'not found.')
    if counter[HTTPStatus.error]:
        plural = 's' if counter[HTTPStatus.error] != 1 else ''
        print('{} error{}'.format(counter[HTTPStatus.error], plural))
    print('Elapsed time: {:.2f}s'.format(elapsed))

def expand_cc_args(every_cc, all_cc, cc_args, limit):
    codes = set()
    A_Z = string.ascii_uppercase
    if every_cc:
        codes.update(a+b for a in A_Z for b in A_Z)
    elif all_cc:
        with open(COUNTRY_CODES_FILE) as fp:
            text = fp.read()
            codes.update(text.split())
    else:
        for cc in (c.upper() for c in cc_args):
            if len(cc) == 1 and cc in A_Z:
                codes.update(cc+c for c in A_Z)
            elif len(cc) == 2 and all(c in A_Z for c in cc):
                codes.add(cc)
            else:
                msg = 'each CC argument must be A to Z or AA to ZZ.'
                raise ValueError('*** Usage error: '+msg)
    return sorted(codes)[:limit]

def process_args(default_concur_req):
    server_options = ', '.join(sorted(SERVERS))
    parser = argparse.ArgumentParser(
        description='Download flags for country codes. '
        'Default: top 20 countries by population.')
    parser.add_argument('cc', metavar='CC', nargs='*',
        help='country code or 1st letter (eg. B for BA...BZ)')
    parser.add_argument('-a', '--all', action='store_true',
        help='get all available flags (AD to ZW)')

```

```

parser.add_argument('-e', '--every', action='store_true',
                    help='get flags for every possible code (AA...ZZ)')
parser.add_argument('-l', '--limit', metavar='N', type=int,
                    help='limit to N first codes', default=sys.maxsize)
parser.add_argument('-m', '--max_req', metavar='CONCURRENT', type=int,
                    default=default_concur_req,
                    help='maximum concurrent requests (default={})'
                        .format(default_concur_req))
parser.add_argument('-s', '--server', metavar='LABEL',
                    default=DEFAULT_SERVER,
                    help='Server to hit; one of {} (default={})'
                        .format(server_options, DEFAULT_SERVER))
parser.add_argument('-v', '--verbose', action='store_true',
                    help='output detailed progress info')
args = parser.parse_args()
if args.max_req < 1:
    print('*** Usage error: --max_req CONCURRENT must be >= 1')
    parser.print_usage()
    sys.exit(1)
if args.limit < 1:
    print('*** Usage error: --limit N must be >= 1')
    parser.print_usage()
    sys.exit(1)
args.server = args.server.upper()
if args.server not in SERVERS:
    print('*** Usage error: --server LABEL must be one of',
          server_options)
    parser.print_usage()
    sys.exit(1)
try:
    cc_list = expand_cc_args(args.every, args.all, args.cc, args.limit)
except ValueError as exc:
    print(exc.args[0])
    parser.print_usage()
    sys.exit(1)
if not cc_list:
    cc_list = sorted(POP20_CC)
return args, cc_list

```



```
def main(download_many, default_concur_req, max_concur_req):
    args, cc_list = process_args(default_concur_req)
    actual_req = min(args.max_req, max_concur_req, len(cc_list))
    initial_report(cc_list, actual_req, args.server)
    base_url = SERVERS[args.server]
    t0 = time.time()
    counter = download_many(cc_list, base_url, args.verbose, actual_req)
    assert sum(counter.values()) == len(cc_list), \
        'some downloads are unaccounted for'
    final_report(cc_list, counter, t0)
```

O script *flags2_sequential.py* (Exemplo A.11) é a base para comparação com as implementações concorrentes. *flags2_threadpool.py* (Exemplo 17.14) também usa as funções *get_flag* e *download_one* de *flags2_sequential.py*.

Exemplo A.11 – flags2_sequential.py

"""Download de bandeiras de países (com tratamento de erros).

Versão sequencial

Exemplo de execução::

```
$ python3 flags2_sequential.py -s DELAY b
DELAY site: http://localhost:8002/flags
Searching for 26 flags: from BA to BZ
1 concurrent connection will be used.
.....
17 flags downloaded.
9 not found.
Elapsed time: 13.36s
```

"""

```
import collections
```

```
import requests
```

```
import tqdm
```

```
from flags2_common import main, save_flag, HTTPStatus, Result
```

```
DEFAULT_CONCUR_REQ = 1
```

```
MAX_CONCUR_REQ = 1
```

```
# BEGIN FLAGS2_BASIC_HTTP_FUNCTIONS
```

```
def get_flag(base_url, cc):
```

```
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
```

```
    resp = requests.get(url)
```

```

    if resp.status_code != 200:
        resp.raise_for_status()
    return resp.content

def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc:
        res = exc.response
        if res.status_code == 404:
            status = HTTPStatus.not_found
            msg = 'not found'
        else:
            raise
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose:
        print(cc, msg)

    return Result(status, cc)

# END FLAGS2_BASIC_HTTP_FUNCTIONS

# BEGIN FLAGS2_DOWNLOAD_MANY_SEQUENTIAL
def download_many(cc_list, base_url, verbose, max_req):
    counter = collections.Counter()
    cc_iter = sorted(cc_list)
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter)
    for cc in cc_iter:
        try:
            res = download_one(cc, base_url, verbose)
        except requests.exceptions.HTTPError as exc:
            error_msg = 'HTTP error {res.status_code} - {res.reason}'
            error_msg = error_msg.format(res=exc.response)
        except requests.exceptions.ConnectionError as exc:
            error_msg = 'Connection error'
        else:
            error_msg = ''
            status = res.status

```

```

        if error_msg:
            status = HTTPStatus.error
            counter[status] += 1
            if verbose and error_msg:
                print('*** Error for {}: {}'.format(cc, error_msg))

    return counter
# END FLAGS2_DOWNLOAD_MANY_SEQUENTIAL

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)

```

Capítulo 19: Scripts e testes para agenda da OSCON

O exemplo A.12 mostra o script de teste para o módulo *schedule1.py* (Exemplo 199). Usa *py.test*, que é uma biblioteca para execução de testes.

Exemplo A.12 – test_schedule1.py

```

import shelve
import pytest

import schedule1 as schedule

@pytest.yield_fixture
def db():
    with shelve.open(schedule.DB_NAME) as the_db:
        if schedule.CONFERENCE not in the_db:
            schedule.load_db(the_db)
        yield the_db

def test_record_class():
    rec = schedule.Record(span=99, eggs=12)
    assert rec.span == 99
    assert rec.eggs == 12

def test_conference_record(db):
    assert schedule.CONFERENCE in db

def test_speaker_record(db):
    speaker = db['speaker.3471']
    assert speaker.name == 'Anna Martelli Ravenscroft'

```

```
def test_event_record(db):
    event = db['event.33950']
    assert event.name == 'There *Will* Be Bugs'

def test_event_venue(db):
    event = db['event.33950']
    assert event.venue_serial == 1449
```

O exemplo A.13 mostra a listagem completa do exemplo *schedule2.py* apresentado na seção “Recuperação de registros relacionados usando propriedades” na página 658 em quatro partes.

Exemplo A.13 – schedule2.py

```
"""
schedule2.py: percorrendo os dados da agenda da OSCON

>>> import shelve
>>> db = shelve.open(DB_NAME)
>>> if CONFERENCE not in db: load_db(db)

# BEGIN SCHEDULE2_DEMO

>>> DbRecord.set_db(db)
>>> event = DbRecord.fetch('event.33950')
>>> event
<Event 'There *Will* Be Bugs'>
>>> event.venue
<DbRecord serial='venue.1449'>
>>> event.venue.name
'Portland 251'
>>> for spkr in event.speakers:
...     print('{0.serial}: {0.name}'.format(spkr))
...
speaker.3471: Anna Martelli Ravenscroft
speaker.5199: Alex Martelli

# END SCHEDULE2_DEMO

>>> db.close()

"""

# BEGIN SCHEDULE2_RECORD
import warnings
import inspect
import osconfeed
```

Apêndice A ■ Scripts auxiliares

```
DB_NAME = 'data/schedule2_db'
```

```
CONFERENCE = 'conference.115'
```

```
class Record:
```

```
    def __init__(self, **kwargs):
```

```
        self.__dict__.update(kwargs)
```

```
    def __eq__(self, other):
```

```
        if isinstance(other, Record):
```

```
            return self.__dict__ == other.__dict__
```

```
        else:
```

```
            return NotImplemented
```

```
# END SCHEDULE2_RECORD
```

```
# BEGIN SCHEDULE2_DBRECORD
```

```
class MissingDatabaseError(RuntimeError):
```

```
    """Levantada quando um banco de dados é necessário, mas não foi definido."""
```

```
class DbRecord(Record):
```

```
    __db = None
```

```
    @staticmethod
```

```
    def set_db(db):
```

```
        DbRecord.__db = db
```

```
    @staticmethod
```

```
    def get_db():
```

```
        return DbRecord.__db
```

```
    @classmethod
```

```
    def fetch(cls, ident):
```

```
        db = cls.get_db()
```

```
        try:
```

```
            return db[ident]
```

```
        except TypeError:
```

```
            if db is None:
```

```
                msg = "database not set; call '{}.set_db(my_db)'"
```

```
                raise MissingDatabaseError(msg.format(cls.__name__))
```

```
            else: # ❶
```

```
                raise
```

```
    def __repr__(self):
```

```
        if hasattr(self, 'serial'):
```

```
            cls_name = self.__class__.__name__
```

```
            return '<{} serial={!r}>'.format(cls_name, self.serial)
```

```

        else:
            return super().__repr__()
# END SCHEDULE2_DBRECORD

# BEGIN SCHEDULE2_EVENT
class Event(DbRecord):

    @property
    def venue(self):
        key = 'venue.{}'.format(self.venue_serial)
        return self.__class__.fetch(key)

    @property
    def speakers(self):
        if not hasattr(self, '_speaker_objs'):
            spkr_serials = self.__dict__['speakers']
            fetch = self.__class__.fetch
            self._speaker_objs = [fetch('speaker.{}'.format(key))
                                  for key in spkr_serials]

        return self._speaker_objs

    def __repr__(self):
        if hasattr(self, 'name'):
            cls_name = self.__class__.__name__
            return '<{} {}!r>'.format(cls_name, self.name)
        else:
            return super().__repr__()
# END SCHEDULE2_EVENT

# BEGIN SCHEDULE2_LOAD
def load_db(db):
    raw_data = osconfeed.load()
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():
        record_type = collection[:-1]
        cls_name = record_type.capitalize()
        cls = globals().get(cls_name, DbRecord)
        if inspect.isclass(cls) and issubclass(cls, DbRecord):
            factory = cls
        else:
            factory = DbRecord

```

```

        for record in rec_list:
            key = '{}.{}'.format(record_type, record['serial'])
            record['serial'] = key
            db[key] = factory(**record)
# END SCHEDULE2_LOAD

```

O exemplo A.14 foi usado para testar o exemplo A.13 com `py.test`.

Exemplo A.14 – `test_schedule2.py`

```

import shelve
import pytest
import schedule2 as schedule

@pytest.yield_fixture
def db():
    with shelve.open(schedule.DB_NAME) as the_db:
        if schedule.CONFERENCE not in the_db:
            schedule.load_db(the_db)
        yield the_db

def test_record_attr_access():
    rec = schedule.Record(spam=99, eggs=12)
    assert rec.spam == 99
    assert rec.eggs == 12

def test_record_repr():
    rec = schedule.DbRecord(spam=99, eggs=12)
    assert 'DbRecord object at 0x' in repr(rec)
    rec2 = schedule.DbRecord(serial=13)
    assert repr(rec2) == "<DbRecord serial=13>"

def test_conference_record(db):
    assert schedule.CONFERENCE in db

def test_speaker_record(db):
    speaker = db['speaker.3471']
    assert speaker.name == 'Anna Martelli Ravenscroft'

def test_missing_db_exception():
    with pytest.raises(schedule.MissingDatabaseError):
        schedule.DbRecord.fetch('venue.1585')

```

```
def test_dbrecord(db):
    schedule.DbRecord.set_db(db)
    venue = schedule.DbRecord.fetch('venue.1585')
    assert venue.name == 'Exhibit Hall B'

def test_event_record(db):
    event = db['event.33950']
    assert repr(event) == "<Event 'There *Will* Be Bugs'>"

def test_event_venue(db):
    schedule.Event.set_db(db)
    event = db['event.33950']
    assert event.venue_serial == 1449
    assert event.venue == db['venue.1449']
    assert event.venue.name == 'Portland 251'

def test_event_speakers(db):
    schedule.Event.set_db(db)
    event = db['event.33950']
    assert len(event.speakers) == 2
    anna_and_alex = [db['speaker.3471'], db['speaker.5199']]
    assert event.speakers == anna_and_alex

def test_event_no_speakers(db):
    schedule.Event.set_db(db)
    event = db['event.36848']
    assert len(event.speakers) == 0
```


Jargão de Python

Muitos termos aqui, é claro, não são exclusivos de Python, mas particularmente nas definições você poderá encontrar sentidos que são específicos para a comunidade Python.

Veja também o glossário oficial de Python (<https://docs.python.org/3/glossary.html>).

ABC (Abstract Base Class [Classe-Base Abstrata])

Uma classe que não pode ser instanciada e da qual pode-se apenas herdar. ABCs são a maneira de formalizar interfaces em Python. Em vez de herdar de uma ABC, uma classe também pode declarar que atende à interface registrando-se com a ABC para se tornar uma *subclasse virtual*.

ABC (linguagem de programação)

Uma linguagem de programação criada por Leo Geurts, Lambert Meertens e Steven Pemberton. Guido van Rossum, que desenvolveu Python, trabalhou como programador implementando o ambiente de ABC nos anos 80. Estruturação de blocos por indentação, tuplas e dicionários embutidos, desempacotamento de tuplas, a semântica do laço for e tratamento uniforme para todos os tipos de sequência são algumas das características marcantes de Python que vieram de ABC.

apelidamento (aliasing)

Atribuir dois ou mais nomes ao mesmo objeto. Por exemplo, em `a = []`; `b = a`, as variáveis `a` e `b` são apelidos (aliases) para o mesmo objeto lista. O apelidamento ocorre naturalmente o tempo todo em qualquer linguagem em que variáveis armazenem referências a objetos. Para evitar confusão, simplesmente esqueça a ideia de variáveis serem caixas que armazenam objetos (um objeto não pode estar em duas caixas ao mesmo tempo). É melhor pensar em variáveis como etiquetas associadas a objetos (um objeto pode ter mais de uma etiqueta).

argumento

Uma expressão passada para uma função quando ela é chamada. No jargão pythônico, *argumento* e *parâmetro* quase sempre são sinônimos. Veja *parâmetro* para saber mais sobre a distinção e o uso desses termos.

atribuição paralela

Atribuir a diversas variáveis a partir de itens de um iterável usando uma sintaxe como `a, b = [c, d]` – também conhecida como atribuição desestruturante (destructuring assignment). É uma aplicação comum de *desempacotamento de tuplas*.

atributo

Métodos e atributos de dados (chamados de “campos” no linguajar de Java) são todos atributos em Python. Um método é apenas um atributo que, por acaso, é um objeto invocável (normalmente, é uma função, mas não necessariamente).

atributo de armazenagem

Um atributo é uma *instância gerenciada* usada para armazenar o valor de um atributo gerenciado por um *descritor*. Veja também *atributo gerenciado*.

atributo gerenciado

Um atributo público gerenciado por um objeto descritor. Embora o *atributo gerenciado* seja definido na *classe gerenciada*, ele funciona como um atributo de instância (ou seja, normalmente tem um valor por instância, mantido em um *atributo de armazenagem*). Veja *descritor*.

ávido (eager)

Um objeto iterável que cria todos os seus itens de uma só vez. Em Python, uma *list comprehension* é ávida. Compare com *preguiçoso (lazy)*.

BDFL

Benevolent Dictator For Life (Ditador Benevolente Vitalício), apelido de Guido van Rossum, criador da linguagem Python.

BOM

Byte Order Mark (Marca de Ordem de Byte) é uma sequência de bytes que pode estar presente no início de um arquivo codificado com UTF-16. Um BOM é o caractere U+FEFF (ZERO WIDTH NO-BREAK SPACE) codificado para gerar `b'\xfe\xff'` em uma CPU big-endian ou `b'\xff\xfe'` em uma CPU little-endian. Como o caractere U+FFFE não existe em Unicode, a presença desses bytes revela a ordem dos bytes usados na codificação, sem que haja ambiguidade. Apesar de ser redundante, um BOM codificado como `b'\xef\xbb\xbf'` pode ser encontrado em arquivos UTF-8.

CamelCase

A convenção para escrever identificadores juntando palavras com iniciais maiúsculas (por exemplo, `ConnectionRefusedError`). A PEP-8 recomenda que nomes de classe devem ser escritos em CamelCase, mas o conselho não é seguido pela biblioteca-padrão de Python. Veja *snake_case*.

Cheese Shop

O nome original do PyPI (Python Package Index, ou Índice de Pacotes Python – <https://pypi.python.org/pypi>), extraído do esquete de Monty Python sobre uma loja de queijos em que nada está disponível. Quando escrevi este livro, o subdomínio original <https://cheeseshop.python.org> ainda estava funcionando. Veja *PyPI*.

classe

Uma construção do programa que define um tipo novo, com atributos de dados e métodos que especificam possíveis operações nesses dados. Veja *tipo*.

classe gerenciada

Uma classe que usa um objeto descritor para gerenciar um de seus atributos. Veja *descritor*.

classe mixin

Uma classe projetada para outra classe herdar, juntamente com outras classes adicionais, em uma árvore de classes com herança múltipla. Uma classe mixin jamais deve ser instanciada, e uma subclasse concreta de uma classe mixin deve herdar também de outra classe que não seja mixin.

code point (código Unicode)

Um inteiro no intervalo de 0 a 0x10FFFF usado para identificar uma entrada na base de dados de caracteres Unicode. Em Unicode 7.0, menos de 3% dos code points possíveis estão atribuídos a caracteres. Na documentação de Python, o termo pode ser encontrado como uma ou duas palavras. Por exemplo, no capítulo 2, “Built-in Functions” (Funções embutidas, <http://docs.python.org/library/functions.html>) de *Python Library Reference* (Guia de referência à biblioteca de Python), a descrição da função `chr` afirma que aceita um “codepoint” inteiro, enquanto sua função inversa `ord` é descrita como devolvendo um “code point Unicode”.

code smell (código que não cheira bem)

Um padrão de código que sugere que pode haver algo errado com o design de um programa. Por exemplo, o uso excessivo de verificações com `isinstance` com classes concretas é um code smell, pois dificulta estender o programa para lidar com tipos novos no futuro.

codec

(codificador/decodificador). Um módulo com funções para codificar e decodificar, normalmente de `str` para `bytes` e vice-versa, embora Python tenha alguns codecs que façam transformações de `bytes` para `bytes` e de `str` para `str`.

coleção

Termo genérico para estruturas de dados compostas de itens acessíveis individualmente. Algumas coleções podem conter objetos de tipos quaisquer (ver *container*), enquanto outras têm somente objetos de um único tipo atômico (veja *sequência simples*). Tanto `list` quanto `bytes` são coleções, mas `list` é um contêiner e `bytes` é uma sequência simples.

considered harmful (considerado prejudicial)

A carta de Edsger Dijkstra intitulada “Go To Statement Considered Harmful” (Instrução `go to` considerada prejudicial) definiu uma fórmula para títulos de artigos que critiquem alguma técnica de ciência da computação. O artigo “*Considered Harmful*” (http://en.wikipedia.org/wiki/Considered_harmful) da Wikipedia lista vários exemplos, incluindo “*Considered Harmful Essays Considered Harmful*” (Artigos *Considered Harmful* considerados prejudiciais, <http://meyerweb.com/eric/comment/chech.html>) de Eric A. Meyer.

construtor

Informalmente, o método de instância `__init__` de uma classe é chamado de construtor, pois sua semântica é semelhante àquela de um construtor em Java. No entanto um nome adequado para `__init__` é *inicializador*, pois ele não cria realmente a instância, mas recebe-a como seu argumento `self`. O termo *construtor* descreve melhor o método de classe `__new__`, que Python chama antes de `__init__`, e é responsável por realmente criar uma instância e devolvê-la. Veja *inicializador*.

contêiner

Um objeto que armazena referências a outros objetos. A maioria dos tipos de coleção em Python é composta de contêineres, mas nem todos são. Compare com *sequência simples*, que são coleções, mas não são contêineres.

cópia profunda (deep copy)

Uma cópia de um objeto em que todos os objetos que são atributos ou subitens do objeto também são copiados. Compare com *cópia rasa* (*shallow copy*).

cópia rasa (shallow copy)

Uma cópia de um objeto que compartilha referências a todos os objetos que são atributos ou subitens do objeto original. Compare com *cópia profunda (deep copy)*. Veja também *apelidamento (aliasing)*.

corrotina

Um gerador usado para programação concorrente, que recebe valores de um escalonador ou de um laço de eventos por meio de `coro.send(value)`. O termo pode ser usado para descrever a função geradora ou o objeto gerador obtido ao chamar a função geradora. Veja *gerador*.

CPython

O interpretador Python padrão implementado em C. Esse termo é usado somente na discussão de comportamentos específicos de implementações ou quando falamos dos vários interpretadores Python disponíveis, por exemplo, PyPy.

CRUD

Sigla para Create, Read, Update e Delete (Criar, Ler, Atualizar e Apagar) – as quatro funções básicas em qualquer aplicação que armazene registros.

decorador

Um objeto invocável *A* que devolve outro objeto invocável *B* e é chamado no código com a sintaxe `@A` imediatamente antes da definição de um invocável *C*. Quando lê um código como esse, o interpretador Python chama *A(C)* e vincula o *B* resultante à variável anteriormente atribuída a *C*, substituindo efetivamente a definição de *C* por *B*. Se o invocável alvo *C* é uma função, *A* é um decorador de função; se *C* é uma classe, *A* é um decorador de classe.

descritor

Uma classe que implementa um ou mais métodos especiais entre `__get__`, `__set__` ou `__delete__` torna-se um descritor quando uma de suas instâncias é usada como um atributo de classe de outra classe, a *classe gerenciada*. Descritores gerenciam o acesso e a remoção de *atributos gerenciados* na *classe gerenciada*, com frequência armazenando dados nas *instâncias gerenciadas*.

descritor dominante (overriding descriptor)

Um *descritor* que implementa `__set__` e, sendo assim, intercepta tentativas de escrita no *atributo gerenciado* da *instância gerenciada*. Também chamado de descritor de dados (data descriptor) ou enforced descriptor (descritor impositivo). Compare com *descritor não dominante (nonoverriding descriptor)*.

descriptor não dominante (nonoverriding descriptor)

Um *descriptor* que não implementa `__set__` e, sendo assim, não interfere com a escrita em um *atributo gerenciado* na *instância gerenciada*. Consequentemente, se um atributo de mesmo nome receber valor na *instância gerenciada*, ele encobrirá o descriptor nessa instância. Também chamado de descriptor não dados (nondata descriptor) ou shadowable descriptor (descriptor mascarável). Compare com *descriptor dominante* (*overriding descriptor*).

desempacotamento de iteráveis

Um sinônimo moderno e mais exato para *desempacotamento de tuplas*. Veja também *atribuição paralela*.

desempacotamento de tupla

Atribuir itens de um objeto iterável a uma tupla de variáveis (por exemplo, `first, second, third == my_list`). É o termo comumente usado por pythonistas, mas *desempacotamento de iterável* está ganhando força.

desfiguração de nomes (name mangling)

A renomeação automática de atributos privados de `__x` para `_MyClass__x`, realizada pelo interpretador Python em tempo de execução.

docstring

Abreviatura de “documentation string” (string de documentação). Quando a primeira instrução em um módulo, uma classe ou função é uma constante string literal, o interpretador assume que ela seja a *docstring* do objeto que a contém, e a armazena como o atributo `__doc__` desse objeto. Veja também *doctest*.

doctest

Um módulo com funções para fazer parse e executar exemplos incluídos nas docstrings de módulos Python ou em arquivos-texto simples. Também pode ser usado a partir da linha de comando assim:

```
python -m doctest module_with_tests.py
```

DRY

Don't Repeat Yourself (Não se repita) – um princípio de engenharia de software que afirma que “Toda informação deve ter uma representação única, não ambígua e incontestável em um sistema.” Surgiu pela primeira vez no livro *The Pragmatic Programmer* de Andy Hunt e Dave Thomas (Addison-Wesley, 1999).

duck typing

Uma forma de polimorfismo em que funções operam sobre qualquer objeto que implemente os métodos apropriados, independentemente de suas classes ou de declarações explícitas de interface.

dunder

Atalho para pronunciar os nomes dos *métodos especiais* e de atributos escritos com underscores duplos no início e no fim (por exemplo, `__len__` é lido como “dunder len”).

EAFP

Sigla que representa a frase “It’s easier to ask forgiveness than permission” (É mais fácil pedir perdão que permissão), atribuída a Grace Hopper, pioneira em computação, e citada por pythonistas referindo-se a práticas de programação dinâmica como acessar atributos sem testar antes para verificar se existem e então capturar a exceção se for o caso. A docstring da função `hasattr` diz que ela funciona “chamando `getattr(object, name)` e capturando `AttributeError`”.

expressão geradora

Uma expressão entre parênteses que usa a mesma sintaxe de uma *list comprehension*, mas devolve um gerador em vez de devolver uma lista. Uma *expressão geradora* pode ser entendida como uma versão *preguiçosa (lazy)* de uma *list comprehension*. Veja *preguiçoso (lazy)*.

fail-fast (falhar logo)

Uma abordagem de design de sistemas que recomenda que erros devem ser informados o mais cedo possível. Python é mais aderente a esse princípio que a maioria das linguagens dinâmicas. Por exemplo, não existe um valor “indefinido”: variáveis referenciadas antes da inicialização geram um erro e `my_dict[k]` levanta uma exceção se `k` não existir (em oposição a JavaScript). Como outro exemplo, a atribuição paralela por meio de desempacotamento de tuplas em Python só funciona se todos os itens forem explicitamente tratados, enquanto Ruby lida silenciosamente com discrepâncias no número de itens, ignorando itens não usados do lado direito de `=` ou atribuindo `nil` a variáveis extras do lado esquerdo.

falso (falsy)

Qualquer valor `x` para o qual `bool(x)` devolva `False`; Python usa `bool` implicitamente para avaliar objetos em contextos booleanos, por exemplo, a expressão que controla um `if` ou um laço `while`. É o oposto de *verdadeiro (truthy)*.

fatiamiento

Produzir um subconjunto de uma sequência usando a notação de fatias, por exemplo, `my_sequence[2:6]`. O fatiamento normalmente copia dados para produzir um novo objeto; em particular, `my_sequence[:]` cria uma cópia rasa de toda a sequência. Entretanto um objeto `memoryview` pode ser fatiado para produzir uma nova `memoryview` que compartilhe dados com o objeto original.

função

Estritamente falando, é um objeto resultante da avaliação de um bloco `def` ou de uma expressão `lambda`. Informalmente, a palavra *função* é usada para descrever qualquer objeto invocável, como métodos e, às vezes, até classes. A lista oficial de Built-in Functions (Funções embutidas, <http://docs.python.org/library/functions.html>) inclui várias classes embutidas como `dict`, `range` e `str`. Veja também *objeto invocável*.

função de ordem superior

Uma função que aceita outra função como argumento, como `sorted`, `map` e `filter`, ou uma função que devolve uma função como resultado, como fazem os decoradores de Python.

função de primeira classe

Qualquer função que seja um objeto de primeira classe na linguagem (isto é, que possa ser criada em tempo de execução, atribuída a variáveis, passada como argumento e devolvida como resultado de outra função). Funções em Python são funções de primeira classe.

função embutida (BIF, ou Built-in Function)

Uma função incluída no interpretador Python, presente na implementação subjacente da linguagem (ou seja, em C no caso de CPython, em Java no caso de Jython e assim por diante). O termo com frequência refere-se apenas às funções que não precisam ser importadas e estão documentadas no capítulo 2, “Built-in Functions” (Funções embutidas, <https://docs.python.org/2/library/functions.html>) em *The Python Standard Library Reference* (Referência à biblioteca-padrão de Python). Porém módulos embutidos como `sys`, `math`, `re` etc., também contêm funções embutidas.

função genérica

Um grupo de funções projetado para implementar a mesma operação de formas customizadas para diferentes tipos de objeto. Em Python 3.4, o decorador `functools.singledispatch` é a maneira-padrão de criar funções genéricas. Isso é conhecido como multimétodos em outras linguagens.

função geradora

Uma função que tem a palavra reservada `yield` em seu corpo. Quando chamada, uma função geradora devolve um *gerador*.

genexp

Abreviatura de “generator expression” (*expressão geradora*).

gerador

Um iterador criado com uma função geradora ou uma expressão geradora que pode produzir valores sem necessariamente iterar por uma coleção; o exemplo canônico é um gerador que produz a série de Fibonacci, que, por ser infinita, jamais caberia em uma coleção. O termo às vezes é usado para descrever uma *função geradora*, além de referir-se ao objeto que resulta de sua chamada.

gerenciador de contexto (context manager)

Um objeto que implementa os métodos especiais `__enter__` e `__exit__` para uso em um bloco `with`.

GoF (livro da GoF)

Apelido do livro *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995)¹, cujos autores são conhecidos como GoF (Gang of Four, ou Gangue dos Quatro); são eles: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides.

hashable

Um objeto é hashable se tiver os métodos `__hash__` e `__eq__`, com a restrição de que o valor de hash jamais deve mudar, e se `a == b`, então `hash(a) == hash(b)` também deve ser `True`. A maioria dos tipos embutidos imutáveis é hashable, mas uma tupla só será hashable se todos os seus itens também forem hashable.

idiom

“Uma maneira de falar que é natural aos falantes nativos de uma linguagem”, de acordo com o Princeton WordNet.

inicializador

Um nome melhor para o método `__init__` (em vez de *construtor*). Inicializar a instância recebida como `self` é a tarefa de `__init__`. A construção propriamente dita da instância é feita pelo método `__new__`. Veja *construtor*.

¹ N.T.: Edição brasileira publicada com o título “Padrões de projeto” (Bookman).

instância gerenciada

Uma instância de uma *classe gerenciada*. Veja *atributo gerenciado* e *descriptor*.

iterador

Qualquer objeto que implemente o método `__next__` sem argumentos, que devolva o próximo item de uma série ou levante `StopIteration` quando não houver mais itens. Os iteradores de Python também implementam o método `__iter__`, portanto também são *iteráveis*. Iteradores clássicos, de acordo com o padrão de projeto original, devolvem itens de uma coleção. Um *gerador* também é um *iterador*, mas é mais flexível. Veja *gerador*.

iterável

Qualquer objeto a partir do qual a função embutida `iter` pode obter um iterador. Um objeto iterável funciona como fonte de itens em laços `for`, comprehensions e desempacotamento de tuplas. Objetos que implementem um método `__iter__` que devolva um *iterador* são iteráveis. Sequências são sempre iteráveis; outros objetos que implementem um método `__getitem__` também podem ser iteráveis.

KISS principle (princípio KISS)

A sigla quer dizer “Keep It Simple, Stupid” (Mantenha a simplicidade, estúpido). Quer dizer procurar a solução mais simples possível, com o menor número de componentes. A frase foi criada por Kelly Johnson, um engenheiro aeroespacial extremamente produtivo que trabalhou na verdadeira Área 51 projetando algumas das aeronaves mais sofisticadas do século 20.

list comprehension

Uma expressão entre colchetes que usa as palavras reservadas `for` e `in` para criar uma lista processando e filtrando elementos de um ou mais iteráveis. Uma *list comprehension* funciona avidamente. Veja *ávido* (*eager*).

listcomp

Abreviatura de *list comprehension*.

metaclasses

Uma classe cujas instâncias são classes. Por padrão, classes Python são instâncias de `type`, por exemplo, `type(int)` é a classe `type`, portanto `type` é a metaclasses. Meta-classes definidas pelo usuário podem ser criadas na forma de subclasses de `type`.

metaprogramação

A prática de escrever programas que usam informações sobre si mesmos para mudar sua estrutura e seu comportamento em tempo de execução. Por exemplo, um ORM pode fazer introspecção em declarações de classes de modelo para determinar como validar campos de registros do banco de dados e converter tipos do banco de dados em tipos Python.

método de acesso

Um método implementado para oferecer acesso a um único atributo de dado. Alguns autores usam *método de acesso* como um termo genérico que inclui métodos getter e setter, enquanto outros o usam para referir-se apenas a getters, chamando os setters de mutators.

método dunder

Veja *dunder* e *métodos especiais*.

método especial

Um método com um nome especial como `__getitem__`, com underscores duplos no início e no fim. Quase todos os métodos especiais reconhecidos por Python estão descritos no capítulo “Data model” (Modelo de dados, <http://bit.ly/1GsZwss>) de *The Python Language Reference* (Guia de referência à linguagem Python), mas alguns usados somente em contextos específicos estão em outras partes da documentação. Por exemplo, o método `__missing__` de mapeamentos é mencionado na seção “4.10. Mapping Types — dict” (Tipos de mapeamento – dict, <http://bit.ly/1QS9Ong>) em *The Python Standard Library* (Biblioteca-Padrão de Python).

método mágico

Mesmo que *método especial*.

método mixin

Uma implementação concreta de um método oferecido em uma ABC ou em uma *classe mixin*.

método não vinculado (unbound method)

Um método de instância acessado diretamente em uma classe não está vinculado a uma instância, portanto dizemos que é um “método não vinculado”. Para ser bem-sucedida, uma chamada a um método não vinculado deve passar explicitamente uma instância da classe como primeiro argumento. Essa instância será atribuída ao argumento `self` do método. Veja *método vinculado (bound method)*.

método vinculado (bound method)

Um método acessado por meio de uma instância torna-se vinculado a essa instância. Qualquer método, na verdade, é um descritor, e quando é acessado, devolve a si mesmo encapsulado em um objeto que vincula o método à instância. Esse objeto é o método vinculado. Ele pode ser chamado sem que o valor de `self` seja passado. Por exemplo, dada a atribuição `my_method = my_obj.method`, o método vinculado pode ser chamado posteriormente como `my_method()`. Compare com *método não vinculado (unbound method)*.

monkey patching

Alterar dinamicamente um módulo, uma classe ou uma função em tempo de execução, normalmente para acrescentar recursos ou corrigir bugs. Como é feito em memória e não por mudança no código-fonte, um monkey patch afeta apenas a instância do programa em execução no momento. Monkey patches quebram o encapsulamento e tendem a ter alto acoplamento com os detalhes de implementação das unidades de código em que está o patch, portanto são vistos como soluções alternativas temporárias, e não como uma técnica recomendável para integração de código.

mutator

Veja *método de acesso*.

objeto arquivo ou similar (file-like object)

Usado informalmente na documentação oficial para referir-se a objetos que implementam o protocolo de arquivos, com métodos como `read`, `write`, `close` etc. Variantes comuns são arquivos-texto contendo strings codificadas com leitura e escrita baseadas em linhas; instâncias de `StringIO`, que são arquivos-texto em memória; e arquivos binários contendo bytes não codificados. Esses últimos podem ser com ou sem buffer. As ABCs dos tipos padrões de arquivos estão definidas no módulo `io` a partir de Python 2.6.

objeto bytes ou similar (bytes-like object)

Uma sequência genérica de bytes. Os tipos `bytes` ou similares mais comuns são `bytes`, `bytearray` e `memoryview`, mas outros objetos que suportem o protocolo de buffer de baixo nível de CPython também são classificados como desse tipo se os seus elementos forem apenas bytes.

objeto invocável (callable)

Um objeto que pode ser chamado com o operador de chamada () para devolver um resultado ou executar alguma ação. Há sete variantes de objetos invocáveis em Python: funções definidas pelo usuário, funções embutidas, métodos embutidos, métodos de instância, funções geradoras, classes e instâncias de classes que implementam o método especial `__call__`.

ORM

Object-Relational Mapper (Mapeamento Objeto-Relacional) – uma API que oferece acesso a tabelas e registros de banco de dados como classes e objetos Python, fornecendo chamadas de métodos para realizar operações de banco de dados. SQLAlchemy é um ORM popular e independente para Python; os frameworks Django e Web2py têm seus próprios ORMs incluídos.

parâmetro

Funções são declaradas com zero ou mais “parâmetros formais”, que são variáveis locais sem vinculação. Quando a função é chamada, os *argumentos* ou “parâmetros reais” passados são vinculados a essas variáveis. Neste livro, tentei usar *argumento* para referir-me a um parâmetro passado para uma função e *parâmetro* para um parâmetro formal na declaração da função. No entanto isso nem sempre é viável porque os termos *parâmetro* e *argumento* são usados indistintamente em toda a documentação e na API de Python. Veja *argumento*.

preguiçoso (lazy).

Um objeto iterável que produz itens por demanda. Em Python, geradores são preguiçosos. Compare com *ávido* (*eager*).

prime (preparar)

Chamar `next(coro)` em uma corrotina para fazê-la avançar até sua primeira expressão `yield` para que ela fique pronta para receber valores em chamadas subsequentes de `coro.send(value)`.

princípio do acesso uniforme

Bertrand Meyer, criador da linguagem Eiffel, escreveu o seguinte: “Todos os serviços oferecidos por um módulo devem estar disponíveis por meio de uma notação uniforme, que não revele o fato de estarem implementados por armazenagem ou processamento.” Propriedades e descritores permitem a implementação do princípio de acesso uniforme em Python. A ausência de um operador `new`, fazendo com que chamadas de função e instanciação de objetos pareçam iguais, é outra manifestação desse princípio: quem chama (o caller) não precisa saber se o objeto invocado é uma classe, uma função ou outro invocável qualquer.

PyPI

O Python Package Index (Índice de Pacotes Python, <https://pypi.python.org/>), onde mais de 60 mil pacotes estão disponíveis; é também conhecido como *Cheese shop* (ver *Cheese shop*). PyPI pronuncia-se como “pai-pi-ai” para evitar confusão com *PyPy*.

PyPy

Uma implementação alternativa da linguagem de programação Python que usa um conjunto de ferramentas para compilar um subconjunto de Python para código de máquina, de modo que o código-fonte do interpretador é, na verdade, escrito em Python. PyPy também inclui um JIT para gerar código de máquina para programas de usuário durante a execução – como faz a VM de Java. Em novembro de 2014, PyPy era 6,8 vezes mais rápido que CPython, em média, de acordo com benchmarks publicados (<http://speed.pypy.org/>). PyPI é pronunciado como “pai-pai” para evitar confusão com *PyPI*.

pythônico

Usado para elogiar código Python idiomático que faça bom uso de recursos da linguagem de modo a ser conciso, legível e, frequentemente, mais rápido também. Usado igualmente para APIs que permitem escrever código que pareçam naturais para programadores Python proficientes. Veja *idiom*.

refcount

O contador de referências que cada objeto de CPython mantém internamente para determinar quando esse objeto pode ser destruído pelo coletor de lixo.

referência forte

Uma referência que mantém um objeto vivo em Python. Compare com *referência fraca*.

referência fraca

Um tipo especial de referência a objetos que não incrementa o contador de referências do objeto *referente*. Referências fracas são criadas com uma das funções e estruturas de dados do módulo *weakref*.

referente

O objeto-alvo de uma referência. Esse termo é mais frequentemente usado para discutir *referências fracas*.

REPL (Read-Eval-Print-Loop, Laço-Ler-Avaliar-Exibir)

Um console interativo, como o python padrão ou alternativas como *ipython*, *bpython* e *Python Anywhere*.

sequência

Nome genérico para qualquer estrutura de dados iterável com um tamanho conhecido (via `len(s)`), que permita o acesso a itens por meio de índices começando em 0 (por exemplo, `s[0]`). A palavra *sequência* faz parte do jargão de Python desde o início, mas somente em Python 2.6 foi formalizada como uma classe abstrata em `collections.abc.sequence`.

sequência binária

Termo genérico para tipos de sequência com elementos do tipo bytes. Os tipos embutidos de sequência binária são: `byte`, `bytearray` e `memoryview`.

sequência simples

Um tipo de sequência que armazena fisicamente os valores de seus itens, e não referências a outros objetos. Os tipos embutidos `str`, `bytes`, `bytearray`, `memoryview` e `array.array` são sequências simples. Compare com `list`, `tuple` e `collections.deque`, que são sequências contêiner. Veja *container*.

serialização

Conversão de um objeto de sua estrutura em memória para um formato binário ou baseado em texto para armazenagem ou transmissão, permitindo a futura reconstrução de um clone do objeto no mesmo sistema ou em um sistema diferente. O módulo `pickle` trata serialização de objetos Python arbitrários para um formato binário.

singleton

Um objeto que é a única instância existente de uma classe – normalmente não por acaso, mas porque a classe é projetada para impedir a criação de mais de uma instância. Há também um padrão de projeto chamado Singleton, que é uma receita para implementar classes desse tipo. O objeto `None` é um singleton em Python.

snake_case

A convenção de escrever identificadores juntando palavras com o caractere underscore (`_`) – por exemplo, `run_until_complete`. A PEP-8 chama esse estilo de “letras minúsculas com palavras separadas por underscores” e recomenda-o para nomear funções, métodos, argumentos e variáveis. Para pacotes, a PEP-8 recomenda concatenar palavras sem usar separadores. A biblioteca-padrão de Python tem vários exemplos de identificadores com `snake_case`, mas também muitos exemplos de identificadores sem separação entre as palavras (por exemplo, `getattr`, `classmethod`, `isinstance`, `str.endswith` etc.). Veja *CamelCase*.

string de bytes (byte string)

Um nome infeliz ainda usado para referir-se a bytes ou bytearray em Python 3. Em Python 2, o tipo `str` era realmente uma string de bytes e o termo fazia sentido para distinguir `str` de strings `unicode`. Em Python 3, não faz sentido insistir nesse termo, e tentei usar *sequência de bytes* sempre que precisei falar em geral sobre... sequências de bytes.

subclasse virtual

Uma classe que não herda de uma superclasse, mas é registrada com `TheSuperClass.register(TheSubClass)`. Veja a documentação de `abc.ABCMeta.register` (<http://bit.ly/1DeDbKf>).

tempo de importação

O momento na execução inicial de um módulo em que seu código é carregado pelo interpretador Python, avaliado do início ao fim e compilado em bytecode. É nesse momento que classes e funções são definidas e tornam-se objetos vivos. É também o instante em que os decoradores são executados.

tipo

Cada categoria específica de dados do programa, definida por um conjunto de valores possíveis e operações sobre esses valores. Alguns tipos em Python são próximos aos tipos de dados de máquina (por exemplo, `float` e `bytes`), enquanto outros são extensões (por exemplo, `int` não está limitado pelo tamanho de palavra da CPU e `str` armazena códigos Unicode de vários bytes) ou abstrações de nível bem alto (por exemplo, `dict`, `deque` etc.). Tipos podem ser definidos pelo usuário por meio de classes, ou podem estar embutidos no interpretador (um tipo “embutido”). Antes da unificação de tipo/classe em Python 2.2, tipos e classes eram entidades diferentes, e classes definidas pelo usuário não podiam estender tipos embutidos. Desde então, tipos embutidos e classes new-style tornaram-se compatíveis e uma classe é uma instância de `type`. Em Python 3, todas as classes são new-style. Veja *classe* e *metaclass*.

user-defined (definido pelo usuário)

Na documentação de Python, quase sempre a palavra *usuário* refere-se a você e eu – programadores que usam a linguagem Python – em oposição aos desenvolvedores que implementam um interpretador Python. Portanto o termo “classe definida pelo usuário” quer dizer uma classe escrita em Python, em oposição às classes embutidas escritas em C, como `str`.

verdadeiro (truthy)

Qualquer valor *x* para o qual `bool(x)` devolva `True`; Python usa `bool` implicitamente para avaliar objetos em contextos booleanos, por exemplo, a expressão que controla um `if` ou um laço `while`. É o oposto de *falso (falsy)*.

view (visão)

As views de Python 3 são estruturas de dados especiais devolvidas pelos métodos `.keys()`, `.values()` e `.items()` de `dict`; elas oferecem uma view dinâmica para as chaves e os valores de `dict` sem duplicação de dados, diferente de Python 2, em que esses métodos devolvem listas. Todas as views de `dict` são iteráveis e aceitam o operador `in`. Além disso, se os itens referenciados pela view forem todos hashable, a view também implementará a interface `collections.abc.Set`. É o caso de todas as views devolvidas pelo método `.keys()` e das views devolvidas por `.items()` quando os valores também são hashable.

vivacidade (liveness)

Um sistema assíncrono, com threads ou distribuído exibe a propriedade de vivacidade quando “algo bom acaba acontecendo” (isto é, mesmo que algum processamento esperado não esteja acontecendo nesse momento, ele será futuramente concluído). Quando ocorre deadlock em um sistema, ele perde sua vivacidade.

wart (verruga)

Um recurso deselegante da linguagem. O famoso post “Python warts” de Andrew Kuchling foi reconhecido pelo *BDFL* como influente na decisão de não oferecer retrocompatibilidade no design de Python 3, pois a maioria das falhas não poderia ser corrigida de outra maneira. Muitos dos problemas citados por Kuchling foram corrigidos em Python 3.

YAGNI

“You Ain’t Gonna Need It” (Você não vai precisar disso), um slogan para evitar a implementação de funcionalidades que não são imediatamente necessárias, baseadas em pressupostos sobre requisitos futuros.

Zen de Python

Digite `import this` em qualquer console de Python a partir da versão 2.2.

Sobre o autor

Luciano Ramalho já era desenvolvedor Web antes do IPO da Netscape em 1995, e passou de Perl para Java e depois para Python em 1998. Desde então, tem trabalhado em alguns dos maiores portais de notícias no Brasil usando Python e ministrando cursos de desenvolvimento web em Python para empresas de mídia, bancos e o Governo Federal. Seu currículo como palestrante inclui PyCon US (2013), OSCON (2002, 2013, 2014) e quinze palestras em vários anos na Python Brasil (a PyCon brasileira) e na FISL (a maior conferência FLOSS do hemisfério sul). Ramalho é *fellow* da Python Software Foundation e cofundador do Garoa Hacker Clube, o primeiro hackerspace do Brasil. É sócio da empresa de treinamento Python.pro.br (<http://python.pro.br>).

SOBRE O AUTOR



Luciano Ramalho

<https://twitter.com/ramalhoorg>

<https://www.linkedin.com/in/lucianoramalho>

<https://github.com/ramalho>

O animal na capa de *Python fluente* é o lagarto da areia de Namaqua (*Pedioplanis namaquensis*) – uma criatura delgada, com uma cauda longa e marrom com tons rosados. É preto com quatro listras brancas, tem patas marrons com pontos brancos e a barriga branca.

Ativo durante o dia, é um dos lagartos mais rápidos que existem. Habita planícies de areia e cascalho, com vegetação esparsa, e hiberna em tocas cavadas na base de arbustos durante o inverno. O lagarto da areia de Namaqua pode ser encontrado em toda a Namíbia, em savanas áridas e regiões semidesérticas. Esse animal alimenta-se de pequenos insetos. Em novembro, as fêmeas põem de três a cinco ovos.

Muitos dos animais nas capas dos livros da O'Reilly estão ameaçados; todos eles são importantes para o mundo. Para saber como você pode ajudar, acesse animals.oreilly.com.

A imagem da capa foi extraída do livro *Wood's Natural History, volume 3*.

editora

Conheça o site da novatec

The image shows a screenshot of the Novatec website with several callouts pointing to different sections:

- Download de conteúdos exclusivos**: Points to the 'Download' link in the top navigation bar.
- Informações, dúvidas, opiniões Fale conosco!**: Points to the 'Fale conosco' link in the top navigation bar.
- Fique conectado pelas redes sociais**: Points to the social media icons in the top right corner.
- Acompanhe nossos lançamentos**: Points to the 'Últimos Lançamentos' section, which features book covers for titles like 'UML', 'Android', 'Java', 'PHP', and 'Python Cookbook'.
- Confira artigos publicados por nossos autores**: Points to the 'Artigos Recentes' section, which lists recent articles with their titles and authors.
- Fique ligado nas principais notícias**: Points to the 'Notícias' section, which lists recent news items.
- Anote na agenda**: Points to the 'Próximos Eventos' section, which lists upcoming events.
- Conheça nossas parcerias**: Points to the 'Parcerias' section, which features logos of partner companies.
- Aguarde os próximos lançamentos**: Points to the 'Próximos Lançamentos' section, which shows upcoming book releases.

www.novatec.com.br

Cadastre seu e-mail e receba mais informações sobre os nossos lançamentos e promoções



A simplicidade de Python permite que você se torne produtivo rapidamente, porém isso muitas vezes significa que você não estará usando tudo que ela tem a oferecer. Com este guia prático, você aprenderá a escrever um código Python eficiente e idiomático aproveitando seus melhores recursos – alguns deles, pouco conhecidos. O autor Luciano Ramalho apresenta os recursos essenciais da linguagem e bibliotecas de Python mostrando como você pode tornar o seu código mais conciso, mais rápido e mais legível ao mesmo tempo.

Muitos programadores experientes tentam dobrar o Python para que ele se enquadre em padrões aprendidos com outras linguagens e jamais descobrem os recursos do Python que estão além de sua experiência. Com este livro, esses programadores Python aprenderão a ser totalmente proficientes em Python 3.

Este livro inclui:

- **O modelo de dados do Python:** entenda como os métodos especiais são o segredo para o comportamento consistente dos objetos.
- **Estruturas de dados:** tire total proveito dos tipos embutidos e entenda a dualidade entre texto e bytes na era do Unicode.
- **Funções como objetos:** veja as funções Python como objetos de primeira classe e entenda como isso afeta alguns padrões de projeto populares.
- **Técnicas de orientação a objetos:** crie classes após dominar referências, mutabilidade, interfaces, sobrecarga de operadores e herança múltipla.
- **Controle de fluxo:** tire proveito de gerenciadores de contexto, geradores, corrotinas e concorrência com os pacotes `concurrent.futures` e `asyncio`.
- **Metaprogramação:** entenda como funcionam propriedades, descritores de atributos, decoradores de classe e metaclasses.

“Tenho orgulho de ter sido um dos revisores técnicos deste livro excelente – a obra não só ajudará muitos programadores Python de nível intermediário em sua jornada para dominar a linguagem como também me ensinou muito!”

—Alex Martelli

Python Software Foundation Fellow

“*Python Fluente* é um baú de tesouros cheio de truques úteis de programação para programadores Python de níveis intermediário e avançado que queiram ampliar as fronteiras de seu conhecimento.”

—Daniel e Audrey Roy Greenfeld

Autores de *Two Scoops of Django*

Luciano Ramalho, é programador Python desde 1998, Fellow da Python Software Foundation, é sócio do `Python.pro.br` – uma empresa de treinamento – e cofundador do Garoa Hacker Clube, o primeiro hackerspace do Brasil. Tem liderado equipes de desenvolvimento de software e ministrado cursos sobre Python em empresas de mídia, bancos e para o governo federal.

Fique conectado:



twitter.com/novateceditora



facebook.com/novatec



www.novatec.com.br

ISBN 978-85-7522-462-5



9 788575 224625